MASTER EN SCIENCES INFORMATIQUES

Améliorations à Chord pour la fourniture de services peer-to-peer sur Internet

François, Pierre

Award date:
2003

Facultés Universitaires Notre-Dame de la Paix
Namur
Institut d'Informatique

Améliorations à Chord pour la fourniture de services peer-to-peer sur Internet

Pierre François

Mémoire présenté en vue
de l'obtention du grade
de Maître en Informatique

Promoteur
Professeur O. Bonaventure

Année académique 2002-2003

Résumé A l'heure où l'engouement pour les systèmes Peer-to-Peer n'est plus à prouver, les faiblesses des systèmes actuels ont motivé le développement de nouveaux modèles Peer-to-Peer, appelés DHT (Distributed Hash Tables). Il semble cependant que la cohérence de ce type de solutions est extrêmement sensible aux comportements frauduleux des hôtes qui constituent ces systèmes. Dans ce mémoire, nous montrons comment l'inaccessibilité d'un hôte par ses pairs, phénomène lié au déploiement de mécanismes de sécurité sur l'Internet, constitue un frein potentiel au déploiement de ce genre d'applications. Pour illustrer nos propos, nous analysons l'impact théorique de l'inaccessibilité des noeuds sur la cohérence d'une DHT particulière : Chord. Ensuite, nous proposons une solution visant à conserver la cohérence de Chord lorsque ce système est déployé sur une topologie "hostile". Enfin, nous évaluons par simulation l'impact de cette inaccessibilité sur Chord et nous montrons que notre solution lève cette contrainte d'inaccessibilité, et ce, en préservant la performance de Chord.

Abstract As the success of Peer-to-Peer applications is no more to prove, new Peer-to-Peer models have arised to counter the scaling problems of the present ones. They are called "Distributed Hash Tables". Even though they are very performant, they seem to be very sensible to potentialy undesired behaviours of the hosts that compose them. In this thesis, we show how unreachable nodes can affect those systems consistency. In that way, we analyse the impact of a node's unreachability on a specific DHT called Chord. Furthermore, we provide a mechanism which aim is to preserve consistency when it is deployed on a "hostile" network topology. After that, we analyse our simulation results to illustrate the impact of the nodes unreachability on Chord. We also show in our simulations that the provided mechanism can counter that problem, while preserving performances.

Je tiens à remercier mon promoteur, le professeur Olivier Bonaventure, pour son extrême disponibilité, ses précieux conseils et les moyens qu'il a mis à ma disposition pour réaliser les simulations.

Je tiens également à remercier Peter Van Roy pour son accueil lors de mon stage. Je voudrais exprimer ma reconnaissance à Bruno Carton et à Valentin Mesaros pour l'attention qu'ils ont portée à mon travail lors de mon stage et tout au long de la rédaction de mon mémoire. Je tiens également à les remercier pour leur conseils sur Oz, sans lesquels la programmation du simulateur aurait été un véritable calvaire.

Je tiens également à remercier les personnes pré-citées pour la liberté de recherche qu'ils m'ont accordée durant mon stage et lors de la réalisation de mon mémoire.

Un grand merci à Pierre Reinbold pour sa disponibilité et ses commentaires pertinents sur la rédaction de mon mémoire. Je tiens également à remercier Bruno Quoitin et Louis Swinnen d'avoir tout fait pour me placer dans un environnement idéal lors de la réalisation des simulations. Encore un merci à Louis pour ses remarques "anti-stress".

Merci à Fredéric et à Catherine de s'être intéressés à mes questions de probabilités.

Je tiens à remercier Bruno et Olivier pour leurs "instants détentes" si nécessaires.

Comment ne pas remercier mes parents pour le soutien moral et fincancier qu'ils m'ont apporté tout au long de mon cursus.

Un grand merci à Aurore pour toutes ces années de bonheur et de complicité passées en sa compagnie.

Enfin, je remercie Isabelle pour sa précieuse aide dans la rédaction de mon mémoire, tant au niveau du fond que de la forme, pour ses petits plats, et pour son adaptation à mon horaire "décalé". Je tiens surtout à la remercier de m'avoir tout simplement supporté.

Table des matières

1 Vers de nouveaux modèles P2P **15**

1.1 Introduction 15

1.2 Faiblesses des systèmes actuels 16

1.2.1 Le modèle "flooded requests" 17

1.2.2 Le modèle de répertoire centralisé 19

1.3 Une solution : les "Distributed Hash Tables" 20

1.3.1 Besoins 20

1.3.2 Un modèle général de Distributed Hash Table 21

1.3.3 Efficacité réelle et marketing 23

1.4 Implémentations du modèle 26

1.4.1 CAN 26

1.4.2 Pastry 28

1.4.3 Chord 31

1.5 Conclusion 36

2 Variantes d'implémentation de Chord **39**

2.1 Modes de propagation des requêtes 39

2.1.1 Le mode itératif 40

2.1.2 Le mode récursif 42

2.1.3 Le mode récursif avec retour direct du résultat 44

2.2 Modes d'envoi d'un message 45

2.3 Maintien de connexions TCP/IP 46

2.4 Conclusion 46

3 Impact de l'invisibilité d'un noeud sur Chord **49**

3.1 Une hypothèse forte sur la couche réseau inférieure 50

3.2 Impact sur le succès de l'insertion d'un noeud dans le système 51

3.2.1 Par établissement systématique de connexion 51
3.2.2 Par utilisation de connexions pré-établies 52
3.3 Impact sur la stabilité du système 54
3.3.1 Comportement du prédécesseur d'un noeud caché 55
3.3.2 Comportement général du système 57
3.4 Amélioration de l'opérateur CLOSEST_PRECEDING_NODE 58
3.5 Conclusion 59

4 Un mécanisme de proxy pour le contact d'un noeud caché 61
4.1 Idée générale 61
4.2 Principe de fonctionnement 62
4.3 Conservation de la cohérence du protocole 64
4.4 Pseudo-code des primitives Chord 65
4.4.1 Résolution de clé 65
4.4.2 Construction de la table de routage 66
4.5 Voies d'optimisation 67
4.5.1 Calcul distribué de la route la plus courte 67
4.5.2 Trace des lookups 68
4.5.3 Recherche du meilleur chemin de connexion 69
4.5.4 Utilisation du mode de propagation récursif avec retour direct du résultat 70
4.5.5 Détection de routes invalides 70
4.5.6 Détection de boucles dans les routes 70
4.5.7 Utilisation de l'opérateur Closest_preceding_node amélioré 71
4.5.8 Amélioration de la politique de connexion 71
4.6 Conclusion 72

5 Description des simulateurs 73
5.1 Le réseau 74
5.1.1 Intérêt de la simulation de la couche réseau 74
5.1.2 Primitives de la couche réseau 75
5.2 Le système avec proxy's 77
5.2.1 Caractéristiques d'un noeud 77
5.2.2 Comportement d'un noeud 79
5.2.3 Procédures de stabilisation 86
5.3 Le système avec proxy's "allégé" 88

5.4 Le système sans proxy 89
5.5 Conclusion 90

6 Simulations **91**
6.1 Contexte des simulations 91
6.2 Résultats 93
6.3 Travaux futurs 96
6.4 Conclusion 97

A Code du simulateur **105**

Introduction

Ces dernières années ont vu naître un ensemble de protocoles qui ont boulversé la perception des usages possibles des technologies liées à l'Internet. Ces protocoles, communément qualifiés de protocoles Peer-to-Peer, s'appuyent sur la coopération d'hôtes, jouant tous le même rôle au sein d'un système distribué, afin de rendre un service décentralisé et dénué de toute forme de contrôle.

L'explosion de l'utilisation de Napster [1] a refleté ce boulversement, tant il a rassemblé les ordinateurs personnels de leurs utilisateurs, formant ainsi un groupe de machines qui coopèrent pour former un système de fichiers gigantesque.

Dans un autre domaine, le projet SETI@home [2], dont le but est l'analyse des signaux perçus par quelques téléscopes radio, balayant le ciel à la recherche d'émissions radio extra-terrestres, a montré que les systèmes Peer-to-Peer permettaient également de regrouper les puissances de calcul (même faibles) des ordinateurs d'une communauté pour former un "super-calculateur" virtuel. Ce projet avait été motivé par l'idée que le processeur des ordinateurs personnels de chacun s'avère, en général, sous-employé.

Enfin, le succès de Gnutella [3], basé sur un système de diffusion massive de messages, a prouvé à chacun qu'il était possible de créer un système permettant la coopération entre des hôtes, et ce, sans recourir à une quelconque forme de centralisation ou de contrôle.

D'autre part, cette expansion des moyens de coopération a été accompagnée d'un besoin accru de sécurité matérialisé dans le déploiement intensif de technologies visant à restreindre l'accès potentiel aux machines "connectées".

Ainsi, la généralisation de l'utilisation du "firewall" (voir [4]), permettant à tous les utilisateurs d'un réseau local de se connecter aux machines situées à l'extérieur de ce réseau, mais interdisant l'accès depuis l'extérieur aux machines qui le composent, boulverse la perception usuelle de l'Internet comme un réseau "symétrique".

Dans le même ordre d'idées, l'usage du mécanisme de "Network Adress Translator" [5], autorisant les membres d'un réseau local d'accéder aux machines d'autres réseaux

sans leur imposer de détenir une adresse IP publique, remet en question l'idée que toute machine du réseau est identifiable, et donc joignable.

Ces derniers temps, les systèmes Peer-to-Peer ont été critiqués, au delà des considérations d'ordre juridique, pour leur consommation de bande passante, souci majeur des fournisseurs d'accès à Internet, et pour leur incapacité potentielle à supporter un nombre de pairs de plus en plus important.

Pour lutter contre les faiblesses supposées des modèles Peer-to-Peer actuels, le monde scientifique a proposé un ensemble de protocoles performants. Ainsi, des protocoles comme Can [6], Pastry [7], Kademlia [8] ou encore Chord [9] ont vu le jour. Leurs similarités les ont conduits à être classés dans un type de modèle P2P appelé DHT (Distributed Hash Table).

Ces protocoles sont totalement décentralisés, chaque pair se voit affecter les mêmes responsabilités et doit accomplir des tâches visant au maintient de la cohérence du système. La nature de ces tâches nous force de constater que, si ces protocoles sont réellement plus performants, ils sont beaucoup plus sensibles aux comportements frauduleux des pairs d'un système.

Or, lorsqu'un membre d'un système Peer-to-Peer n'est pas accessible, en raison de l'utilisation des techniques de firewall ou de NAT, nous pouvons considérer qu'il a un comportement frauduleux, en ce sens qu'il ne permet pas à ses pairs de l'accéder et donc de coopérer avec lui pour accomplir les tâches de maintien de cohérence du système.

Pourtant, il semble que les "auteurs" de DHT n'ont pas, à l'heure actuelle, porté d'attention particulière à ce genre de phénomènes. Il semble donc nécessaire, avant d'envisager le déploiement effectif de DHT sur Internet, d'en évaluer la correction et les performances en intégrant le problème de l'inaccessibilité potentielle des noeuds.

Ce mémoire est, dans cette optique, destiné à l'analyse de l'impact de ce facteur sur la cohérence des systèmes Peer-to-Peer de type "DHT".

Pour réaliser cette analyse, nous proposons, dans le chapitre 1, un modèle général de DHT afin d'établir une connaissance des caractéristiques communes de ce type de système. Nous décrivons également trois protocoles particuliers (Can, Pastry et Chord), dans les termes de notre modèle général.

Nous nous concentrons ensuite sur Chord en proposant différentes possibilités d'implémentation respectueuses de sa définition théorique. Ceci constituera le deuxième chapitre du mémoire.

Le chapitre 3 discute de l'impact de l'inaccessibilité des noeuds d'un système Chord

sur sa cohérence et sa stabilité, en fonction des différentes variantes d'implémentation proposées au chapitre 2. Sur base de cette discussion, nous observons que les services rendus par Chord sont compromis par l'inaccessibilité potentielle des pairs du système et nous identifions les caractéristiques d'implémentations de Chord qui ont tendance à minimiser cet impact.

Au vu de ces observations, nous proposons, dans le chapitre 4, une extension à Chord permettant de contrer cet effet. Une version "allégée" de cette extension sera également proposée. Cette dernière permet d'assurer la cohérence de Chord tout en limitant la quantité d'informations à maintenir, au sein de chaque noeud, pour permettre son application. Nous montrons que notre solution n'altère pas les performances de Chord lorsqu'elle est utilisée dans un environnement où elle n'est pas nécessaire, c.à.d. lorsque tous les membres du système sont accessibles.

Enfin, les deux derniers chapitres sont consacrés, d'une part, à la description d'un simulateur permettant d'étudier le comportement d'un système Chord déployé sur un réseau "contraignant" ainsi que le comportement d'un système fonctionnant selon l'extension proposée et sa variante "allégée" et, d'autre part, à la description du contexte des simulations effectuées ainsi qu'à l'étude des résultats de ces simulations. Nous observons, grâce à ces simulations, que notre extension répond aux objectifs pour lesquels elle était destinée. Nous montrons que les contextes d'environnement créés pour effectuer les simulations empêchent toutefois de réaliser une comparaison exhaustive des performances de notre solution avec sa version allégée.

Chapitre 1

Vers de nouveaux modèles P2P

Dans ce chapitre, nous expliquons les faiblesses majeures des systèmes Peer-to-Peer actuels. Nous montrons en quoi ces faiblesses peuvent compromettre l'évolutivité quantitative de ces systèmes. Nous en déduisons une série d'exigences pour l'élaboration d'un système Peer-to-Peer performant et "scalable".

Nous introduisons ensuite la notion de table de hachage distribuée (DHT) en proposant un modèle général pour celle-ci. Nous en déduisons le potentiel en terme de performance d'un système Peer-to-Peer basé sur ce principe.

Sur base de ce modèle général, nous discutons la pertinence d'une comparaison de performance des DHT avec les systèmes existants. Pour ce faire, nous montrons que les services proposés par les DHT dans leurs versions actuelles sont différents des services rendus par des applications telles que Gnutella ou Napster. Nous proposons une voie d'adaptation des DHT visant à rendre ces services identiques sans en compromettre la performance.

Enfin, nous présentons quelques solutions existantes en montrant de quelle manière celles-ci répondent à notre modèle général.

1.1 Introduction

La multiplication incessante des machines personnelles connectées à l'Internet et l'accroissement constant des volumes de stockage de celles-ci ont contribué au succès d'applications telles que Napster et Gnutella. Au delà du battage médiatique réalisé autour des problèmes juridiques liés au partage d'oeuvres d'art digitalisées, des doutes ont été émis,

ces dernières années, au sujet de la capacité de ces systèmes à s'adapter à leur évolution en terme de nombre de clients. Certains ISP (Internet Service Provider) ont également émis des craintes au sujet de la quantité de trafic engendré par les clients Gnutella sur leur réseaux. C'est dans ce contexte que de nouveaux modèles Peer-to-Peer, censés remédier à ces problèmes de performance, ont vu le jour. Ces nouveaux modèles disposent, pour la plupart, de caractéristiques communes dans leur mode de fonctionnement. Cet aspect a poussé la communauté P2P à taxer ces modèles du nom de DHT, pour "Distributed Hash Table", tant leur interface est semblable à une table de hachage dont le contenu est distribué entre plusieurs hôtes.

Il semble que le monde scientifique a répondu aux "appels de détresse" des ISP en proposant des modèles théoriques performants. Toutefois, l'applicabilité de ces modèles peut paraître compromise au vu de certains aspects critiques des DHT. Si les auteurs de DHT ne semblent pas destiner leur modèle à des applications de partage de fichiers personnels (nous expliquerons pourquoi cela nous semble irréalisable), mais plutôt à des applications distribuées comme les implémentations du modèle de "publish/subscribe", le "file mirroring" ou encore les systèmes de fichiers distribués, leur but est d'aboutir à des applications dont le fonctionnement serait distribué sur Internet. Cet aspect ajoute, par rapport aux applications distribuées sur un réseau local, un nombre importants de contraintes supplémentaires auxquelles les scientifiques ne semblent pas avoir porté une grande attention.

Dans ce chapitre, nous commençons par justifier brièvement les craintes émises au sujet de la "scalabilité" des systèmes actuels. Ensuite, nous proposons un modèle général de table de hachage. Nous explicitons, sur base de ce modèle, les aspects des DHT qui constituent des faiblesses potentielles pour les implémentations qui en découlent. Nous mettons en évidence les aspects contraignants pour l'implémentation d'un "file-sharing system" basé sur une DHT. Nous tentons ensuite d'identifier quelques adaptations permettant de réduire l'impact de ces contraintes. Enfin, nous présentons le fonctionnement théorique de Can, Pastry et, finalement, de Chord, en expliquant comment ils correspondent à notre modèle de table de hachage distribuée. Cette présentation permettra de réaliser la performance et la fragilité potentielle de ce type de systèmes.

1.2 Faiblesses des systèmes actuels

Dans cette section, nous explicitons les principales justifications des craintes émises au sujet des systèmes P2P actuels, en critiquant les modèles sur lesquels ils reposent. Pour ce faire, nous discutons du modèle de flooded requests sur lequel est basé Gnutella et du modèle de répertoire centralisé, principe de fonctionnement du système Peer-to-Peer

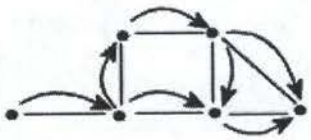

FIG. 1.1 – Propagation d'une requête selon le modèle de "flooded requests".

hybride Napster.

1.2.1 Le modèle "flooded requests"

En se penchant sur le trafic généré par les clients Gnutella et sur le principe de fonctionnement du mécanisme de recherche dont disposent ces clients, on peut se rendre compte que l'accroissement du trafic induit par l'arrivée de nouveaux noeuds, et donc de nouveaux émetteurs de requêtes, est trop important pour que l'on puisse considérer le système Gnutella comme "scalable".

Lorsqu'un hôte connecté au système effectue une requête de recherche, il l'envoie à tous les hôtes qui lui sont voisins. Ceux-ci propagent à leur tour cette requête selon la même méthode. Ainsi, lorsqu'un hôte reçoit une requête, il la transmet à tous les hôtes auxquels il est connecté sauf à celui qui lui a envoyé la requête. Si, dans le graphe représentant la topologie d'un réseau gnutella, il existe plusieurs chemins entre deux noeuds, on peut constater qu'une requête passant par un de ces noeuds empruntera tous les chemins possibles pour arriver à l'autre noeud.

Un mécanisme fonctionnant sur base d'un champ "Time to live" similaire au champ TTL des paquets IP est utilisé. Lors de chaque retransmission, la valeur de ce champ est décrémentée et un paquet ayant une valeur de champ TTL nulle n'est plus retransmis. Ce mécanisme constitue un moyen de limitation de portée des requêtes afin de ne pas saturer le réseau. On peut remarquer que cela implique le non déterminisme du système.

Bien que ce mécanisme de TTL constitue un moyen de protection contre le bouclage des requêtes (c'est d'ailleurs la fonction principale du champ TTL des paquets IP), une méthode plus efficace, en terme de consommation de bande passante, a été implantée dans

Gnutella. La méthode est très simple : lorsqu'un noeud initie une requête, il l'identifie grâce à une valeur numérique. Lors de chaque traitement de requête par un noeud, celui-ci enregistre l'identifiant de la requête dans une cache. Avant de retransmettre une requête à ses pairs, le noeud vérifie simplement, sur base de cette cache, qu'il ne l'a pas déjà retransmise auparavant. Si la requête a déjà été retransmise, elle est ignorée.

Nous pouvons remarquer que cette solution est mieux adaptée pour Gnutella étant donné que la réception d'une requête déjà reçue ne constitue pas un indice d'erreur de configuration du système (comme la réception d'un paquet déjà retransmis sur un réseau IP). En effet, on peut comprendre, au vu de la méthode de propagation des requêtes, que ce genre de phénomène est assez fréquent et totalement prévu au sein de ce type de modèle.

Notons que l'algorithme d'identification des requêtes est tel que la probabilité que deux noeuds émettent une requête avec le même identifiant est extrêmement faible. Cet aspect combiné avec le champ TTL des requêtes assure le non bouclage de celles-ci et ce, avec une efficacité reconnue.

Le nombre de clients joignables croît géométriquement avec la valeur du champ TTL spécifiée dans les paquets Gnutella. Le nombre de connexions que maintient un client jouit de la même influence sur l'accessibilité. Tout utilisateur cherchant à maximiser ses capacités de recherche dispose donc de deux éléments puissants lui permettant d'arriver à ses fins. Si le nombre d'hôtes joignables augmente géométriquement avec ces deux paramètres, le trafic engendré augmente de la même façon.

Jordan Ritter, dans [10], illustre ces observations en supposant que tous les utilisateurs disposent des mêmes paramètres et qu'une requête nécessite un paquet IP de 83 bytes. Le premier tableau montre l'importance du nombre de connexions et du TTL des requêtes pour l'accessibilité des hôtes. Le second montre l'impact de ces paramètres sur le trafic engendré par une requête. Notons que Jordan Ritter est un des principaux créateurs du système Napster. Ses dires semblent toutefois confirmés par la plupart des recherches sur le sujet comme, par exemple, [11].

Nombre maximum d'utilisateurs joignables (T : *TTL* N : *Nombre de connexions*)

	$T=1$	$T=2$	$T=3$	$T=4$	$T=5$	$T=6$	$T=7$	$T=8$
$N=2$	2	4	6	8	10	12	14	16
$N=3$	3	9	21	45	93	189	381	765
$N=4$	4	16	52	160	484	1.456	4.372	13.120
$N=5$	5	25	105	425	1.705	6.825	27.305	109.225
$N=6$	6	36	186	936	4.686	23.436	117.186	585.936
$N=7$	7	49	301	1.813	10.885	65.317	391.909	2.351.461
$N=8$	8	64	456	3.200	22.408	156.864	1.098.056	7.686.400

Trafic généré (en Bytes) (T : TTL N : Nombre de connexions)

	$T=1$	$T=2$	$T=3$	$T=4$	$T=5$	$T=6$	$T=7$	$T=8$
$N=2$	166	332	498	664	830	996	1.162	1.328
$N=3$	249	747	1.743	3.735	7.719	15.687	31.623	63.495
$N=4$	332	1.328	4.316	13.280	40.172	120.848	362.876	1.088.960
$N=5$	415	2.075	8.715	35.275	141.515	566.475	2.266.315	9.065.675
$N=6$	498	2.988	15.438	77.688	388.938	1.945.188	9.726.438	48.632.688
$N=7$	581	4.067	24.983	150.479	903.455	5.421.311	32.528.447	195.171.263
$N=8$	664	5.312	37.848	265.600	1.859.864	13.019.712	91.138.648	637.971.200

Chaque hôte dispose donc d'un pouvoir sensible sur le fonctionnement du système et sur sa capacité à supporter l'arrivée de nouveaux noeuds. En effet, si les clients ont tendance à maximiser le champ TTL et le nombre de connexions qu'ils peuvent établir, on peut imaginer que le système risque, à terme, de s'étouffer dans la masse de trafic qu'il engendre.

En outre, ce type de mécanisme ne permet de fournir une borne raisonnable ni pour le temps de résolution d'une requête ni pour le nombre de pairs nécessaires à l'atteinte de la ressource recherchée. Le système est indéterministe en ce sens qu'un hôte n'est pas assuré qu'une réponse positive à une requête lui sera retournée, même si un hôte tiers présent dans le système est en mesure de la lui fournir. Cela signifie, dans le cas particulier de Gnutella, qu'un noeud effectuant une requête n'est pas sûr de recevoir une réponse même si un ou plusieurs noeuds présents à ce moment dans le système disposent de fichiers répondant aux critères spécifiés dans la requête.

1.2.2 Le modèle de répertoire centralisé

Napster constitue le plus connu des répertoires centralisés, son comportement est décrit, par exemple, dans [1]. Le débat sur l'opportunité de la classification de ce modèle dans les modèles P2P nous conduit à considérer ce modèle comme un système P2P hybride. Selon ce modèle, les membres du système se connectent à un répertoire commun où ils enregistrent les ressources dont ils disposent. Lors d'une demande d'accès à une ressource, un noeud contacte le répertoire et fournit les critères correspondant à la ressource recherchée. Le répertoire parcourt les enregistrements qu'il contient et élit le ou les pairs susceptibles de fournir la réponse la plus adéquate aux critères de recherche. Le noeud demandeur peut ainsi contacter de manière directe le ou les noeuds qui partagent la ressource.

Le fait que toute l'information concernant les ressources soit concentrée en un point semble constituer un défaut majeur du modèle en terme de "scalabilité". En effet, le serveur qui répertorie les ressources doit supporter une bande passante plus importante

au fur et à mesure que le nombre de requêtes par intervalle de temps croît. De plus, la capacité de stockage de ce même serveur doit également augmenter de façon linéaire par rapport à l'augmentation du nombre de ressources partagées. Ceci tend à rendre plus ardue l'évolution quantitative de tels systèmes.

1.3 Une solution : les "Distributed Hash Tables"

Les craintes émises dans la section précédente ont motivé l'élaboration de modèles de systèmes Peer-to-Peer susceptibles de fournir un service ne souffrant pas des défauts supposés des applications actuelles. Dans cette section, nous présentons quelques exigences reconnues comme fondamentales en vue de l'élaboration de nouveaux systèmes Peer-to-Peer.

Partant de la constatation que la plupart des prototypes proposés actuellement peuvent être vus comme des implémentations distribuées d'une table de hachage, nous en proposons ensuite un modèle général.

Enfin, nous expliquons pourquoi nous pensons que les performances de ce type de système (dans leurs versions actuelles) ne peuvent être comparées avec les performances de systèmes tels que Gnutella.

1.3.1 Besoins

Le développement de systèmes Peer-to-Peer déterministes, efficaces en terme de trafic engendré et robustes vis à vis de l'évolution du nombre d'hôtes présents semble donc intéressant au vu de l'engouement incessant pour les systèmes Peer-to-Peer et des faiblesses des implémentations actuelles.

Ces systèmes doivent en outre être conçus de manière à limiter l'influence d'un noeud sur leur comportement global. Il ne faut en aucun cas qu'un noeud puisse bloquer le système, de manière volontaire ou non.

Si nous excluons la possibilité d'utiliser des services reposant sur des répertoires centralisés ou sur une diffusion massive de messages (Broadcast), nous devons penser les systèmes Peer-to-Peer comme des infrastructures où l'information sur la localisation des ressources est distribuée de telle façon que chaque pair du système ne doive maintenir qu'une faible quantité d'information de routage. La recherche d'une ressource s'exécute alors de façon incrémentale par le transfert de proche en proche de la requête correspon-

dante, en dirigeant progressivement celle-ci vers le ou les pairs les mieux "informés" pour y répondre.

Le recours à la distribution de l'information de routage entraîne l'incohérence ponctuelle du système lorsque son environnement change (lors des départs et arrivées de pairs dans le système). Les pairs d'un tel système doivent donc continuellement s'adapter aux modifications de leur environnement. Il s'avère, dès lors, également souhaitable qu'un nombre restreint de noeuds voient leur information de routage influencée par une telle modification de l'environnement.

1.3.2 Un modèle général de Distributed Hash Table

Diverses solutions ont été apportées dans le but de remédier aux différents inconvénients des systèmes P2P actuels. Les auteurs de Chord [9], CAN [6], Pastry [7] et Tapestry [12] font reposer l'intérêt de leurs systèmes sur la décentralisation, la "scalabilité" et la performance des requêtes de recherche en terme de nombre d'hôtes contactés et de trafic engendré. La complexité (en terme de nombre de sauts sur le réseau Peer-to-Peer) de l'opération de résolution d'une requête est de l'ordre de $\log(N)$, N représentant le nombre de noeuds dans le système. Le volume d'information de routage maintenu par les pairs de tels systèmes est également de l'ordre de $\log(N)$.

Une analyse de ces différents modèles permet de constater qu'ils fonctionnent tous selon des principes similaires. Dans [13], les auteurs identifient déjà ces principes en classant ces mécanismes dans l'ensemble des implémentations du "Document routing model". Il semble cependant que la communauté P2P taxe ces nouveaux modèles du nom de DHT (Distributed Hash Table). Effectivement, il convient de constater que l'interface de ces systèmes offre des fonctionnalités similaires aux tables de hachage classiques.

Nous tentons donc d'identifier ces principes communs en proposant un modèle général de DHT. Les 5 points suivants décrivent un ensemble de mécanismes permettant à un groupe d'ordinateurs connectés via un réseau de rendre un service identique à celui d'une table de hachage ; le contenu de celle-ci étant distribué parmis les constituants du groupe, appelés hôtes ou noeuds.

Identification des noeuds et des ressources

Chaque hôte se voit affecté un identifiant numérique calculé, par exemple, sur base d'une fonction de hachage appliquée à son adresse IP. A chaque document ou ressource partagée correspond également un identifiant numérique (sur base d'une fonction de hachage appliquée à son contenu ou à son nom). Les valeurs des identifiants de ressource et

d'hôte se situent dans le même espace que nous désignons, ici, par le symbole V.

Pour un état donné du système, on distingue un sous-ensemble de V, noté N, correspondant à l'ensemble des noeuds participant au système.

Répartition des responsabilités des pairs

Lorsqu'un noeud est présent dans le système, il se voit attribuer la responsabilité d'un certain nombre de ressources. Chaque implémentation du modèle doit définir la manière à laquelle les ressources sont affectées aux différents noeuds du système. Nous pouvons donc affirmer que chaque implémentation du modèle décrit une relation R_{key} injective et surjective, dépendante de l'état du système, qui associe un identifiant de noeud et un identifiant de ressource et est vérifiée lorsque le noeud est effectivement responsable de la ressource.

Cette relation étant injective et surjective, elle admet un inverse fonctionnel et partout défini r_{key} de signature $r_{key} : V \rightarrow N$.

Organisation de l'information de routage

Pour des raisons de "scalabilité", tout noeud d'un système répondant au modèle ne doit maintenir qu'une vision partielle de la topologie du réseau P2P auquel il participe. Ainsi, chaque noeud ne connaît qu'un sous-ensemble des noeuds présents dans le système. Chaque implémentation du modèle doit donc définir une relation R_{node} qui associe un identifiant de noeud à un autre identifiant de noeud et est vérifiée lorsque le second noeud fait partie de la vision partielle du premier de la topologie du réseau. Cette relation est partout définie, irréflexive et telle que le graphe (N, R_{node}) est connexe.

Résolution d'un "lookup"

Un hôte doit pouvoir effectuer une requête d'accès à un document ou à une ressource partagée dans le système. Pour ce faire, il doit connaître la valeur de la clé correspondant à cette ressource[1]. Cette requête de recherche est appelée *lookup*. Le résultat d'un lookup dans le système, pour une clé k, est une référence vers le noeud responsable de clé k, en l'occurence $r_{key}(k)$.

Pour résoudre un *lookup*, un hôte commence par chercher, parmis les hôtes qu'il connaît, celui dont l'identifiant tend le plus à vérifier la relation R_{key} avec la clé correspondant à la ressource recherchée. Il transfère la requête à ce noeud, qui effectue la même

[1] Ceci pose d'ailleurs problème pour le développement d'applications désireuses de profiter des avantages de ce modèle.

opération. La recherche se propage ainsi de noeud en noeud et se termine lorsqu'elle parvient à l'hôte effectivement responsable de la clé[2]. Le noeud émetteur de la requête est alors averti de l'identité du noeud responsable de la clé.

Plus formellement, la résolution d'un lookup initié en n et portant sur la clé k peut être assimilée à la construction d'un chemin de longueur minimale, de n à $r_{key}(k)$, dans le graphe (N, R_{node}). Cette résolution s'appuie sur une fonction de résolution "nexthop" incorporée au sein du système.

En d'autres termes, chaque implémentation du modèle doit définir une fonction de la forme

$$\sigma : N \times V \to N$$

satisfaisant la propriété ci-dessous.

Soient $n \in N$ et $k \in V$. La suite de noeuds $(n_i)_{i \in \mathbb{N}}$, définie par les équations

$$\begin{cases} n_0 = n, \\ \forall i > 0, n_i = \sigma(n_{i-1}, k), \end{cases}$$

stabilise en un certain rang p tel que $(n_0, ..., n_p)$ est un chemin de longueur minimale, de n à $r_{key}(k)$, dans le graphe (N, R_{node}).

Remarquons que l'efficacité d'un système peut notamment être évaluée en analysant la tendance des relations R_{key} et R_{node} à minimiser la longueur des chemins empruntés par les requêtes.

Gestion des départs et arrivées de noeuds

Lors des départs et arrivées de noeuds, le système s'adapte et réaffecte les responsabilités des noeuds pour se maintenir dans un état de cohérence. Pour entrer dans le système, un noeud doit simplement disposer d'un accès à un noeud, quelconque, déjà présent dans celui-ci.

1.3.3 Efficacité réelle et marketing

Dans cette section, nous exprimons quelques remarques visant à nuancer l'apport des solutions de type DHT, en terme de performance et d'applicabilité.

[2]En pratique, la recherche peut se terminer lorsqu'un noeud peut décider de façon certaine de l'identité de ce responsable.

1.3.3.1 Identifiants aléatoires et proximité des hôtes

L'utilisation de valeurs "aléatoires" pour les identifiants de noeuds introduit un inconvénient dont toutes les implémentations risquent de souffrir. En effet, cela implique que la relation "voisin dans le paysage du système P2P" n'a aucun lien avec une quelconque proximité dans la couche réseau sur lequel fonctionne le système. On ne peut donc pas rendre deux noeuds voisins dans le système pour profiter de leur proximité sur le réseau. On peut donc rencontrer des situations où, pour trouver une ressource qui se situe sur une machine proche, un noeud effectue une recherche qui effectuera un trajet important sur le réseau. Les "auteurs" de DHT semblent vouloir contrer cet inconvénient en ajoutant des mécanismes ou en modifiant la méthode d'affectation des identifiants numériques afin de rendre leur implémentation plus attentive à la topologie du réseau sur lequel elle repose. Nous verrons ainsi que les protocoles CAN et Pastry, même s'ils sont des implémentations de DHT, disposent de propriétés les rendant sensibles à la proximité des hôtes.

1.3.3.2 Localité des ressources

Certains auteurs n'hésitent pas à qualifier leur DHT de solution extrêmement performante, en comparaison avec Gnutella. Ainsi, dans [7], on peut lire :

> "Pastry, along with Tapestry, Chord and Can, represent a second generation of peer-to-peer routing and location schemes that were inspired by the pioneering work of systems like FreeNet and Gnutella. Unlike that earlier work, they guarantee a definite answer to a query in a bounded number of network hops, while retaining the scalability of FreeNet and the self-organizing properties of both FreeNet and Gnutella. "

Il semble cependant que les buts avoués des DHT, dans leur définitions actuelles, ne correspondent pas aux services rendus par Gnutella. Gnutella offre un partage de fichiers entre une multitude de particuliers situés sur un réseau "hétérogène", alors que les applications promises des DHT relèvent souvent de la distribution de contenu dans un réseau d'entreprise. Nous allons tenter d'en expliquer la raison principale.

Un aspect critique des DHT réside dans l'imposition de la localité des ressources. Pour illustrer l'impact de cette contrainte, nous pouvons envisager une implémentation de DHT visant à rendre le service le plus populaire des applications P2P : le "file sharing".

De par sa nature même, une telle application impose le contenu informationnel stocké localement par chaque pair. En effet, chaque ressource (en l'occurence chaque fichier) est référencée dans le système grâce à une clé. Chaque pair du système est également identifié

par une valeur numérique obtenue par une opération mathématique. Ces deux aspects d'identification numérique combinés avec la définition par le système de la relation R_{key} impose donc une localisation arbitraire des ressources proposées par le système. L'évolution des volumes de stockage et de la bande passante disponibles chez les particuliers peut nous laisser imaginer que l'utilisateur futur des systèmes de partage de fichiers pourrait être prêt à céder une partie de ses ressources à la "communauté", permettant ainsi le fonctionnement de "notre" application.

Cette application imaginaire nous amène cependant à nous poser certaines questions.

- Nous pouvons arguer sans crainte que les ressources matérielles des pairs sont potentiellement très diversifiées. Comment réagir alors si la relation R_{key} impose la responsabilité de fichiers de taille conséquente à un utilisateur doté de ressources limitées ou d'une bande passante faible ?
- Le départ d'un hôte du système doit être comblé par le transfert des ressources dont il est responsable vers un hôte dont l'identité dépend de la relation R_{key}. Peut-on sérieusement envisager ce transfert dans un contexte d'utilisation par le particulier ? Retenons également que le départ d'un hôte n'est pas forcément volontaire, ceci remettant évidement en question la possibilité réelle d'effectuer ce transfert.
- Une requête de recherche dans le système correspond à un lookup paramétré par la clé identifiante de la ressource souhaitée. Cette aspect est-il compatible avec la méthode de recherche la plus utilisée (parce qu'elle est la plus adaptée) par les utilisateurs, à savoir la spécification de sous-parties du nom de la ressource ? Quels moyens peuvent être mis en place pour fournir à l'utilisateur la connaissance de la clé identifiante de le ressource qu'il recherche ?
- Enfin, dans une toute autre optique, comment répondre à la question de la responsabilité juridique des données stockées par chaque utilisateur dans un tel contexte ? Le monde du Peer-to-Peer n'a sans doute pas besoin de nouveaux conflits avec la justice.

Ces quelques remarques nous font penser que les services proposés par les DHT sont de nature différente des services rendus par Gnutella ou Napster. Posons nous alors la question de savoir quel crédit accorder à une comparaison des performances des DHT (dans leur définition actuelle) avec Gnutella.

Nous pouvons proposer une piste pour lever certaines contraintes liées à la nature des DHT et donc permettre de les adapter à la recherche de documents telle qu'elle est considérée par les utilisateurs. Ainsi, nous pouvons redéfinir le sens attaché à la relation R_{key} de telle façon qu'un pair ne soit pas responsable du maintien des ressources correspondant aux clés avec lesquelles la relation R_{key} est vérifiée mais bien responsable du maintien de **l'information sur la localisation** de ces ressources. Ainsi, lorsqu'un hôte

rentre dans le système, il s'enregistre auprès du responsable de la clé correspondant à chaque ressource qu'il détient localement. Lorsqu'une requête de résolution d'une clé parvient au responsable effectif de cette clé, celui-ci répond en envoyant la liste des hôtes qui se sont enregistrés pour cette clé. Une solution similaire a été proposée pour Kademlia, une DHT décrite dans [8].

1.4 Implémentations du modèle

Dans cette section, nous présentons trois implémentations du modèle de DHT. Pour ce faire, nous décrivons comment ces implémentations respectent, point par point, la définition de notre modèle général de DHT, fournie à la section 1.3.2.

1.4.1 CAN

CAN (Content Adressable Network) est décrit dans [6]. Une de ses principales caractéristiques est la considération de l'espace des identifiants comme un espace cartésien à dim dimensions dont chaque noeud est responsable d'un sous-espace, "dim-rectangulaire".

Identification des noeuds et des ressources

La valeur de hachage de l'adresse IP d'un noeud est interprétée comme les coordonnées d'un point dans un espace cartésien E de dimension dim fixée. A chaque ressource correspond une clé k calculée également au moyen d'une fonction de hachage et interpretée comme les coordonnées d'un point de E. L'espace V du modèle général correspond donc ici à E.

Répartition des responsabilités des pairs

Chaque noeud entrant dans le système se voit affecter une "zone géographique", correspondant à un sous ensemble des points de l'espace E. L'identifiant du noeud représente un point de cette zone géographique. Cette zone est $dim - rectangulaire$. La relation de responsabilité R_{key} associe un noeud N à une clé k et est vérifiée lorsque k appartient à la zone géographique affectée à N.

A tout moment, l'entièreté de l'espace E est affectée aux noeuds du système et l'intersection de deux zones géographiques détenues par deux noeuds différents est vide. Ceci traduit respectivement la surjectivité et l'injectivité de la relation R_{key}.

Organisation de l'information de routage

Deux noeuds vérifient la relation de "connaissance" R_{node} s'ils sont responsables de zones géographiques adjacentes.

Remarquons que, pour ce modèle particulier, la relation R_{node} est symétrique, i.e

$$\forall\, (A, B) \in N^2 \,:\, A\, R_{node}\, B \Rightarrow B\, R_{node}\, A$$

Résolution d'un lookup

Lorsqu'un "lookup" pour une clé k est traité par un hôte, celui-ci transmet, s'il n'est pas responsable de la clé k, la requête vers le noeud dont les coordonnées géographiques sont les plus proches des coordonnées que représente k. De cette façon, la requête est bien transférée de sorte à minimiser la distance restant pour aboutir à $r_{key}(k)$.

La fonction de résolution de nexthop σ correspond donc au calcul, parmis l'ensemble des identifiants des hôtes responsables des zones adjacentes, de l'identifiant tel que la distance euclidienne entre celui-ci et k est minimale. Notons que, lorsque $n = r_{key}(k)$, $\sigma(n, k) = n$.

Le noeud responsable est trouvé lorsque la requête aboutit à un noeud dont la responsabilité couvre le point défini dans la requête. Ce noeud vérifie donc la relation R_{key} avec la clé recherchée.

Gestion des départs et arrivées de noeuds

Lors d'une arrivée d'un nouveau noeud dans le système, le contact d'entrée du noeud effectue un *lookup* avec, comme valeur de clé, les coordonnées du noeud entrant calculées par hachage. Le noeud responsable apprend de cette façon l'arrivée d'un pair et divise sa zone géographique en deux zones. Cette division se fait de manière à minimiser la somme en valeur absolue des différences entre les dimensions de chaque zone. Le but de cette méthode est d'éviter qu'un noeud puisse être responsable de zones très étendues sur une seule dimension et minime sur une autre, afin de réduire le nombre de zones adjacentes. A deux dimensions, ce mécanisme tend à rendre les zones les plus carrées possibles. Le noeud restreint sa responsabilité à une de ces deux parties et donne la responsabilité de l'autre moitié au nouveau noeud. Les responsables des zones adjacentes à ces deux moitiés sont prévenus de la modification afin que ceux-ci mettent à jour leur table de routage.

Lors d'un départ, un mécanisme distribué permet aux différents voisins de se répartir la zone libre. Ceci se déroule de manière à ce que les noeuds responsables des plus petites zones en prennent une part plus importante.

L'avantage majeur de cette solution se situe dans la "paramétrisabilité" de la localisation des noeuds dans l'espace. En effet, la fonction de hachage peut-être utilisée pour obtenir les valeurs des coordonnées d'un point de E restreint à un nombre fixé de dimensions i. Ainsi, les $dim - i$ valeurs restantes peuvent être déterminées sur base d'un ensemble de caractéristiques du noeud.

On peut, par exemple, utiliser la valeur de hachage pour calculer les $dim - 1$ premières coordonnées du point et l'adresse IP du noeud pour calculer la dernière. Ce calcul se fait de manière à rapprocher, sur cette dimension, des machines proches sur le réseau. La proximité de deux machines sur le réseau obtient ainsi une influence sur leurs proximité, dans l'espace cartésien, des zones dont ils sont responsables. L'intérêt de cette méthode réside dans le fait que les noeuds communiquent essentiellement avec leurs voisins.

1.4.2 Pastry

Pastry, dont le fonctionnement est défini dans [7], propose une API permettant l'élaboration d'applications telles que le stockage distribué de fichiers, la communication de groupe ou encore les "naming systems".

Plusieurs applications reposant sur ce type de DHT ont déjà été proposées. Ainsi, PAST [14, 15] propose un système de fichiers distribué basé sur Pastry. SCRIBE [16] constitue une implémentation du modèle "publish/subscribe" utilisant Pastry pour distribuer, au sein d'une communauté, les opérations de souscription et de publication d'évènements.

Pastry constitue un modèle permettant, comme CAN, de "rapprocher" dans le système des hôtes proches dans la topologie du réseau sur lequel ce système repose. Pour ce faire, la résolution d'un *lookup* pour une clé donnée k par un hôte H ne produit pas la référence vers le noeud tendant le plus à vérifier la relation R_{key} avec la clé k. En fait, la résolution d'une requête produit un ensemble E (dont la taille est un paramètre du système) de références vers des noeuds tendant à vérifier cette relation. Ensuite, le noeud H élit, parmis les membres de E, celui auquel il transfèrera effectivement la requête. Cette élection se déroule sur base d'une heuristique assurant qu'un message soit transféré en priorité vers un noeud "proche" de N. Une métrique permettant d'évaluer la proximité entre deux noeuds peut se définir, par exemple, en terme de nombre de sauts IP nécessaires à l'envoi d'un message entre ces deux noeuds.

Nous allons maintenant montrer en quoi Pastry répond à notre modèle général de DHT.

Identification des noeuds et des ressources

Chaque hôte dispose d'un identifiant numérique constitué d'une chaîne de 128 bits, résultat de l'application d'une fonction de hachage (comme SHA1 [17]) sur son adresse IP.

De par le caractère diversifié des applications prévues pour Pastry, la nature des clés n'est pas définie. Elles doivent juste être dispersées dans le même espace que les identifiants de noeud. Il faut donc se placer au niveau "applicatif" pour discuter de la méthode employée pour affecter les clés. Ainsi, pour SCRIBE, chaque clé correspond à un type d'évènement auquel les hôtes peuvent souscrire. Cette clé est calculée en hachant le nom du sujet. Pour PAST, chaque clé est obtenue en hachant le nom du fichier et l'identifiant de son propriétaire. Remarquons que cet aspect générique de Pastry ne relève que de la manière de présenter le modèle. Il ne faut pas en déduire que Pastry permet de réaliser des applications plus diversifiées que ses "concurrents".

Lors du routage, les identifiants de noeud et les clés sont considérés comme des séquences de chiffres en base 2^b, b étant un paramètre du système.

Répartition des responsabilités des pairs

Le responsable d'une clé k est le noeud dont l'identifiant est le plus proche de k. Cette proximité s'évalue sur base de la différence entre l'identifiant du noeud et la valeur de la clé, considérés tous les deux comme des nombres en base 2^b. Il faut supposer qu'une règle permet de décider de la responsabilité d'une clé lorsque les deux noeuds les plus proches de celle-ci se situent à égale distance de cette même clé.

Organisation de l'information de routage

Comme énoncé dans le modèle général, chaque noeud doit maintenir un ensemble de références vers une partie des pairs constitutifs du système. Pastry distingue 3 sous-ensembles de noeuds au sein de cet ensemble de pairs.

1. La "routing table", constituée de $\lceil \log_{2^b} N \rceil$ lignes, comptant $2^b - 1$ entrées chacunes (N étant le nombre maximal de noeuds présents dans le système). Les entrées de la ligne n sont des références vers des noeuds dont les n premiers chiffres de l'identifiant sont communs avec le noeud courant, mais dont le $n+1$ème chiffre diffère du $n+1$ème chiffre de l'identifiant du noeud courant.

   Plusieurs noeuds sont susceptibles de convenir pour une entrée donnée de la table de routage d'un noeud. Lorsqu'un noeud est mis au courant de l'existence de deux pairs appropriés pour une entrée de sa table de routage, il y place celui qui est le plus proche selon la définition de la métrique de proximité du système.

2. Le "neighborhood set" M contient les identifiants et les références (Adresse IP ou Adresse IP et port) des noeuds les plus proches du noeud courant, dans les termes de la métrique de proximité sur le réseau.
3. Le "leaf set" L contient les références des noeuds dont les identifiants numériques sont les plus proches de l'identifiant numérique du noeud courant. Il est en fait constitué de deux ensembles L^+ et L^- de taille $|L|/2$ reprenant les identifiants numériques les plus proches qui sont respectivement supérieurs et inférieurs à l'identifiant du noeud courant.

   On considère 2^b comme valeur par défaut pour $|M|$ et $|L|$.

Notons que la méthode de construction de la routing table rend implicite la considération des critères de proximité dans la politique de routage des messages par les noeuds.

Résolution d'un lookup

La fonction de résolution de nexthop σ est définie de la manière explicitée si dessous.

Lorsqu'un *lookup* est reçu par un noeud N pour une clé k, N consulte L. Si k est comprise entre le minimum et le maximum des identifiants de L, alors la requête est transférée vers le noeud $Next \in L$ tel que $|k - Next|$ est minimal.

Sinon, le message est transmis au noeud $Next$ de la routing table dont l'identifiant dispose d'un préfixe commun avec k plus long que le préfixe commun de l'identifiant du noeud courant et k. Si $Next$ est introuvable au sein de la routing table, alors le message doit être transféré vers un noeud présent dans la routing table ou dans $L \cup M$, disposant d'un préfixe commun avec k au moins aussi long que celui du noeud courant et tel que son identifiant est plus proche de k que l'identifiant du noeud courant.

Gestion des départs et arrivées de noeuds

Les auteurs de Pastry supposent que, lorsqu'un noeud rentre dans le système, il choisit un noeud de contact proche (dans les termes de la métrique de proximité utilisée). Remarquons que la cohérence du système n'est pas compromise si le contact choisi n'est pas un noeud proche du noeud entrant. Cette contrainte additionnelle n'est nécessaire que pour garantir une utilisation "économique" du réseau sur lequel repose le système.

Lorsqu'un noeud d'identifiant X entre dans le système via un noeud de contact d'identifiant A, ce dernier effectue un *lookup* spécial dans le système pour une clé de valeur égale à X. Cette requête aboutit à un noeud Z dont l'identifiant est le plus proche de X. Tous les noeuds qui ont constitué le chemin par lequel ce *lookup* spécial est passé envoient leurs différentes tables (routing table, L, M) à X. X initialise alors ses tables sur base

des tables de ses voisins et prévient les noeuds dont les tables doivent changer suite à son arrivée.

Le départ (volontaire ou non) d'un noeud est géré de manière paresseuse par ses pairs. Etant donné qu'une alternative pour le transfert d'une requête est toujours possible, et ce grâce à la méthode de gestion des requêtes, chaque noeud se contente d'utiliser une alternative lorsqu'il s'aperçoit que le noeud résultat de l'opération de sélection du "next-hop" défini plus haut n'est plus valide. Parallèlement, il exécute une procédure de résolution d'une nouvelle entrée dont le fonctionnement diffère selon la table dans laquelle est enregistré le noeud en échec.

Lorsque le noeud en échec est un noeud repris dans le "leaf set" du noeud courant, ce dernier demande les entrées du "leaf set" du noeud le plus éloigné (numériquement) du noeud courant, et ce dans la direction du noeud en échec. Ainsi, si l'identifiant du noeud en échec a une valeur numérique plus faible que le noeud courant, ce dernier consulte le "leaf set" du noeud ayant l'identifiant le plus faible parmis l'ensemble des noeuds de son propre "leaf set" pour élire un remplaçant pour le noeud en échec.

Si le noeud en échec est un noeud enregistré dans la routing table du noeud courant, celui-ci consulte les tables de routages des noeuds enregistrés sur la même ligne que le noeud en échec pour élire une nouvelle entrée correspondante.

1.4.3 Chord

Chord est, notamment, décrit dans [9]. Une de ses caractéristiques majeures réside dans la considération de l'espace des identifiants comme un espace numérique.

Identification des noeuds et des ressources

Chaque noeud et chaque ressource reçoit un identifiant numérique obtenu par une fonction de hachage (SHA-1) décrite dans [17]. Cette fonction s'applique, pour un noeud, à son addresse IP. Pour une ressource, elle s'applique à son nom ou à son contenu. SHA-1 est supposée rendre minime la probabilité que deux valeurs de hachage obtenues à partir d'adresse IP différentes soient identiques. Elle est supposée répartir ces valeurs de manière uniforme dans un espace fixé. Pour Chord, cet espace est un ensemble de valeurs numériques dont la taille (T) est égale au nombre maximal de noeuds présents dans le système.

L'utilisation d'un espace numérique implique l'existence d'une relation d'ordre (totale) entre les noeuds du système. Cette relation permet de définir la notion de successeur et de prédécesseur d'un noeud.

Répartition des responsabilités des pairs

A un instant donné, un noeud n vérifie la relation R_{key} avec une ressource Res si :

$$Res \in]Predecesseur(n), n]$$

Remarquons que, dans cette formule, la relation d'appartenance et la notion d'intervalle porte sur les identifiants numériques de Res et de n.

On définit le successeur d'un noeud d'identifiant n comme le noeud dont l'identifiant numérique *succ* suit n dans l'espace des identifiants.

$$Successeur(n) = \begin{cases} Min(N) & \text{si } n = Max(N) \\ Min\{n' \in N \mid n < n'\} & \text{sinon} \end{cases}$$

Avec N désignant l'ensemble des identifiants numériques des noeuds présents dans le système.

La notion de prédécesseur utilisée plus haut peut être définie grâce à cette notion de successeur. En effet, le prédécesseur d'un noeud n est le noeud *pred* dont le successeur est n. Il faut remarquer que, comme illustré dans la définition formelle du successeur, la relation de succession s'exprime en considérant l'aspect "cyclique" de l'espace des identifiants. Ainsi, pour $T = 64$, l'intervalle $[60, 3]$ existe et est constitué par les noeuds $\{60, 61, 62, 63, 0, 1, 2, 3\}$. Ainsi, le successeur du noeud 63 est 1 à condition qu'aucun noeud présent dans le système ne porte l'identifiant 0.

Dans la figure 1.2, nous illustrons l'affectation des responsabilités de ressources notées Kx, dont la valeur de clé est x. On remarque l'aspect cyclique de l'espace des identifiants en observant que le noeud $N1$ d'identifiant 1 est le responsable de la ressource d'identifiant 64. En effet, cette valeur de clé est comprise entre 62 et 1, valeurs des identifiants des noeuds les plus proches, de part et d'autre de 1.

Organisation de l'information de routage

Un noeud d'identifiant N connaît les pairs responsables des clés faisant partie de l'ensemble suivant.

$$\{X = (N + 2^{i-1}) \bmod T \mid 1 \leq i \leq log_2(T)\}$$

T étant la taille de l'espace des valeurs d'identifiant de noeud.

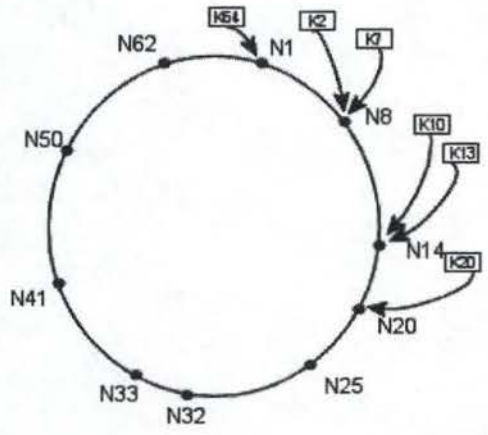

FIG. 1.2 – Les noeuds sont responsables des ressources dont la valeur de clé est comprise entre leur identifiant et l'identifiant de leur prédécesseur.

On a donc la propriété suivante pour Chord :

$$A\, R_{node}\, B \Leftrightarrow \exists i\, :\, 1 \leq i \leq log_2(T) : B\, R_{key}\, ((A + 2^{i-1})\ mod\ T)$$

L'exemple suivant illustre cette propriété : si le nombre maximum de noeuds présents dans le système est 64, alors le noeud dont l'identifiant est 1 connaît les noeuds responsables des clés de l'ensemble $\{2, 3, 5, 9, 17, 33\}$. Ces clés sont obtenues en additionnant la valeur de son identifiant (1) avec le nombre (2^{i-1}) pour la $i^{ème}$ entrée de sa table de routage.

Le tableau ci-après fournit ces "responsables" dans le contexte de la figure 1.3.

Noeud	*Ecart*	*Clé*	*Responsable*
$N1$	1	2	$N8$
$N1$	2	3	$N8$
$N1$	4	5	$N8$
$N1$	8	9	$N14$
$N1$	16	17	$N20$
$N1$	32	33	$N33$

Résolution d'un lookup

Lors d'un "lookup", un noeud transmet la requête au noeud dont la valeur de l'identifiant précède la clé recherchée et s'en approche le plus. Lorsqu'un noeud se trouve mieux

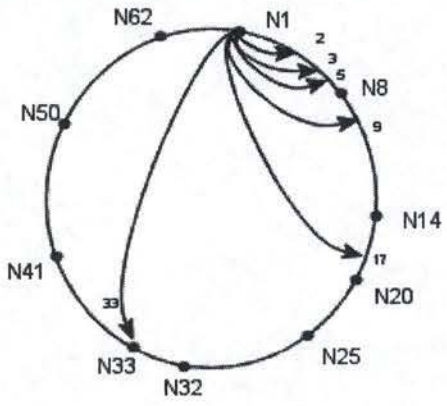

FIG. 1.3 – Ensemble des connaissances de $N1$.

placé que toutes ses connaissances, il désigne son successeur comme étant le responsable effectif de la clé. En effet, si il est lui-même le plus proche noeud qui précède la clé, alors son successeur se situe après cette clé. On peut en déduire que la clé a une valeur comprise entre lui-même et son successeur. Ainsi, le successeur est bien le responsable de la clé.

On notera *Closest_preceding_node* l'opération de recherche du noeud qui précède la clé et s'en approche le plus. Notons que cette opération correspond à la fonction de résolution de nexthop σ du modèle général.

Un exemple de lookup est donné à la figure 1.4. Le noeud $N62$ effectue un lookup pour la clé 31. Conformément à la description de l'organisation de la table de routage d'un noeud, celle-ci contient, pour le noeud $N62$, l'ensemble de noeuds $\{N1, N8, N14, N34\}$. Le noeud dont la valeur d'identifiant précède la clé 31 en s'en approchant le plus est $N14$. $N62$ transmet donc sa requête à $N14$. La table de routage de $N14$ contient les noeuds $\{N20, N25, N32, N50\}$. Le noeud le plus proche qui précède 31 est $N25$. C'est donc à $N25$ que $N14$ retransmet la requête. $N25$ remarque qu'aucun noeud de sa table de routage n'est plus proche de 31 que lui même. Autrement dit, la clé 31 est comprise entre la valeur de son identifiant (25) et l'identifiant de son successeur (32). Par définition, $N32$ est donc le responsable de la clé 31. $N25$ peut alors répondre à l'émetteur initial de la requête ($N62$) en lui fournissant les références vers le noeud $N32$.

Gestion des départs et arrivées de noeuds

Lorsqu'un noeud N rentre dans le système, le contact d'entrée du noeud effectue un

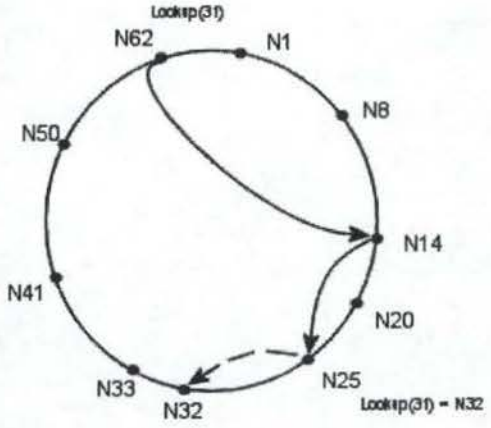

FIG. 1.4 – Exécution théorique d'un *lookup* sur la clé 31 à partir du noeud $N62$.

lookup avec la valeur de l'identifiant du noeud entrant comme valeur de clé. Le résultat correspond au futur successeur du noeud entrant. N contacte alors son successeur pour l'avertir de son arrivée. De cette façon, le successeur peut mettre à jour son prédécesseur, considérant le noeud entrant comme son nouveau prédécesseur.

Pour conserver la cohérence du système, chaque noeud doit maintenir une vue correcte de son successeur. Pour cela, chaque hôte demande périodiquement le prédecesseur de son successeur présumé. S'il s'avère que ce prédécesseur se situe avant le successeur présumé, alors le noeud vient de découvrir un meilleur successeur. Il met à jour sa vue vers celui-ci. Chaque noeud émet périodiquement un message vers son successeur afin que celui-ci puisse mettre à jour sa vue vers son prédécesseur. Dans l'article [9], les auteurs proposent le pseudo-code 1.1 pour la stabilisation.

Lorsqu'un noeud quitte le système, il laisse un de ses pairs sans successeur valide. Pour que le système ne s'effondre pas au fur et à mesure des départs des noeuds, chaque noeud maintient une liste contenant ses successeurs consécutifs. La taille de cette liste doit être établie en fonction du nombre maximal de noeuds admis dans le système et de la probabilité de départs simultanés de noeuds adjacents. Lorsqu'un noeud se rend compte du départ de son successeur, il parcourt la liste, en commençant par le noeud le plus proche, jusqu'à trouver un noeud valide qu'il considère, alors, comme son nouveau successeur.

Un noeud crée sa liste de successeurs au fil de l'évolution du système. Initialement, il ne connaît que son successeur, la liste de ses successeurs ne contient donc qu'une seule

Pseudo-code 1.1 (Stabilisation dans Chord)

```
n.stabilize()
   x=successor.get_predecessor()
   if x between(n,successor)
   then
      successor=x
   end
   successor.notify(n)

n.notify(n2)
   if (predecessor is nil or n2 between(predecessor,n))
   then
      predecessor = n2
   end
```

entrée. Lorsqu'il reçoit un message de notification de son prédécesseur, il lui répond par l'envoi sa liste de successeurs. Lorsqu'un noeud reçoit une liste de successeurs, il lui ajoute son propre successeur, en enlevant le successeur le plus lointain si la liste a déjà atteint sa taille maximale.

Remarquons que cette liste est incohérente si sa taille maximale est supérieure au nombre de noeuds effectivement présents dans le système. Chaque noeud est donc contraint d'élaguer cette liste de sorte qu'elle ne contienne pas de cycle.

1.5 Conclusion

Après avoir énoncé les faiblesses potentielles des modèles P2P de "première génération", nous avons décrit le principe de table de hachage distribuée. Sur base du modèle général de DHT, nous avons identifié certains aspects critiques pour l'implémentation d'applications basées sur ce principe. La manière utilisée pour la description des modèles existants (sur base d'un modèle général) met en évidence la similarité de ce genre de solutions. Il peut donc s'avérer intéressant de traduire les améliorations possibles des protocoles existants dans les termes d'un modèle général de DHT. Ainsi, les "auteurs" de DHT pourront transposer ces améliorations à leur modèle particulier.

Nous avons étudié les responsabilités de chaque pair d'un système P2P basé sur une DHT. Nous pouvons naturellement nous poser, au vu de la répartition des responsabilités au sein du système, la question de savoir comment ces modèles réagissent quand, pour une raison quelconque, certains pairs du système ne respectent pas ces responsabilités. L'idéal consisterait en l'intégration, dans le modèle général lui-même, de solutions permettant de

gérer les comportements potentiellement frauduleux des participants. La complexité de la problématique de gestion des comportements frauduleux au sein d'un système distribué est telle que, lorsque une solution devient robuste (grâce à l'ajout de fonctionnalités) contre un type de comportement frauduleux, l'adaptation de ces solutions dans les termes du modèle général peut, dans les cas où cette adaptation est réalisable, profiter à tous les types de DHT existantes.

Remarquons, pour terminer, que l'apport indéniable des DHT dans l'amélioration des performances des systèmes P2P a été réalisé au prix d'un certain nombre de contraintes que les "développeurs" devront évaluer afin de déterminer si, dans leur cas d'utilisation précis (file sharing, file mirroring, publish/subscribe, etc.), elles n'en rendent pas l'utilisation trop complexe voire impossible.

Chapitre 2

Variantes d'implémentation de Chord

L'article [9] décrit un modèle mathématique correspondant aux spécifications de Chord. Bien entendu, plusieurs implémentations différentes peuvent répondre de manière correcte à ces spécifications. Dans ce chapitre, nous identifions plusieurs caractéristiques clés afin de catégoriser les différentes implémentations possibles du modèle. Nous proposons également les pseudo-codes correspondant à ces variantes d'implémentation.

Cette catégorisation permettra d'étudier l'équivalence effective des implémentations, du point de vue de l'efficacité des requêtes et de la robustesse du système en général.

2.1 Modes de propagation des requêtes

La description de Chord n'impose pas la méthode d'exzcution des requêtes dans le système. Elle définit uniquement une spécification pour les relations que nous avons identifiées, à savoir : R_{key}, R_{node} et l'opération *Closest_preceding_node*. L'algorithme proposé pour le lookup est décrit de manière récursive, mais les auteurs précisent que le protocole est défini par une spécification et non par un algorithme. Ainsi, plusieurs "styles" d'implémentation du lookup peuvent respecter cette spécification.

Nous identifions 3 modes de propagation des requêtes. Ces trois styles d'implémentation respectent la spécification du protocole. Le premier mode, dit "itératif", laisse à l'émetteur d'une requête l'entière responsabilité de la "découverte" du responsable de la clé recherchée. Le second, dit "récursif", fonctionne sur base d'un transfert pur et simple de la requête jusqu'au noeud pouvant y répondre. La méthode utilisée pour renvoyer la réponse à l'émetteur permet de distinguer un troisième et dernier mode de propagation, appelé "mode récursif avec retour direct du résultat". Le mode récursif simple prévoit

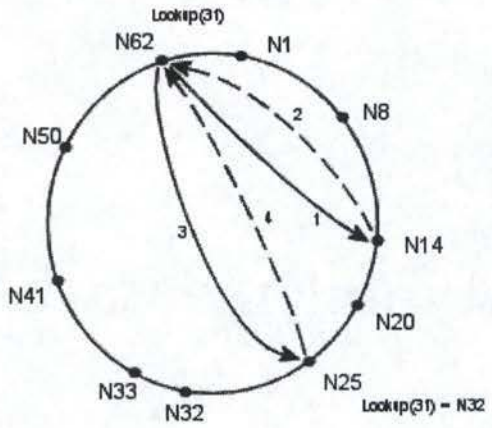

FIG. 2.1 – Exécution itérative d'un "lookup" sur la clé 31 à partir du noeud $N62$.

que la réponse à la requête emprunte le chemin inverse à celui parcouru par la requête, tandis que, selon l'autre mode, la réponse est envoyée directement à l'émetteur initial de la requête.

2.1.1 Le mode itératif

Nous étudions maintenant, de manière plus détaillée, le fonctionnement du mode itératif.

Si un noeud n exécute un lookup pour une clé k donnée, il exécute l'opérateur *Closest_preceding_node* pour k. Si il est le mieux placé, le résultat est trouvé, c'est son successeur. Sinon, il effectue une demande au noeud résultat de l'opération.

Celui-ci applique l'opérateur *Closest_preceding_node* sur la clé. Dans le cas où il est lui-même le résultat de l'opération, il répond à n en envoyant la référence vers son successeur et en spécifiant que ce noeud est le responsable effectif de la clé. Sinon, il envoie le résultat en spécifiant que celui-ci est le noeud le plus proche de la ressource qu'il connaisse.

n effectue la même requête sur chaque noeud qu'il reçoit en réponse tant que celui-ci n'est pas le responsable effectif de la ressource.

Pseudo-code 2.1 (Traitement d'une requête en mode itératif)

```
NextNode=n.Closest_preceding_node(K)
   if (NextNode==n)
   then
      Send(Source,answer(K,successor,true))
   else
      Send(Source,answer(K,NextNode, false)
   end
```

Pseudo-code 2.2 (Recherche d'une clé en mode itératif)

```
n.lookup(Key K)
   NextNode=n.Closest_preceding_node(K)
   if (NextNode==n)
   then
      return(n.successor)
   else
      Responsible=false
      while(not Responsible)
         do
             Send(NextNode,request(K,n.ip))
             Wait(Answer)
            (K,NextNode, Responsible)=(Answer)
         end
      return(NextNode)
   end
```

Lorsqu'un noeud n reçoit un message $request(K, Source)$, il le traite selon le pseudo-code 2.1. Le résultat est un couple reprenant un noeud et un booléen spécifiant si le résultat est le responsable effectif de la clé. Notons qu'un noeud ne doit pas maintenir d'information d'état supplémentaire lorsqu'il reçoit et traite une requête. Quand un noeud effectue une recherche pour une clé K donnée, il exécute la procédure décrite par le pseudo-code 2.2.

Un exemple de lookup itératif est illustré à la figure 2.1. Le noeud $N62$ trouve $N14$ comme "nexthop" pour la requête visant à résoudre le responsable de la clé 31. Il lui envoie le message $request(31, N62.ip)$. $N14$ trouve $N25$ comme nexthop pour cette requête, qui n'est pas le responsable de la clé. $N14$ répond donc à $N62$ en envoyant le message $answer(31, N25, false)$. $N62$ émet alors le message $request(31, N62.ip)$ en direction de $N25$. $N25$ trouve le responsable de la clé 31 ($N32$) et envoie donc le message $answer(31, N32, true)$ à $N62$.

2.1.2 Le mode récursif

Nous décrivons ici le fonctionnement d'une résolution de clé selon le mode récursif.

Si un noeud N exécute un lookup pour une clé K donnée, il exécute l'opération *Closest_preceding_node* pour K. Si il est le mieux placé, le résultat est trouvé, c'est son successeur. Sinon, il effectue un lookup sur le noeud résultat de l'opération. Celui-ci fonctionne de la même manière. Lorsqu'un noeud a trouvé le résultat, donc lorsqu'il découvre que son successeur est le responsable de la clé, il envoie celui-ci au noeud qui l'a contacté pour effectuer le lookup. Le résultat "remonte" ainsi la suite des noeuds qui ont participé au lookup jusqu'à N.

Il existe deux grandes variantes possibles pour effectuer ce type de requête. La première consiste à encoder l'information sur le parcours effectué par la requête dans le message qui lui correspond. Chaque noeud consulte alors cette information pour décider à qui retransmettre la réponse. La seconde méthode réside dans l'établissement d'une cache des requêtes reçues et transférées. De cette façon, lorsqu'un noeud reçoit une réponse, il peut consulter cette cache pour élire le noeud auquel il va retransmettre cette réponse.

Remarquons que la seconde méthode impose aux noeuds de maintenir de l'information d'état pour une durée indéterminée. Il faut alors se poser la question de savoir combien de temps un noeud devra maintenir cette information avant de considérer la requête comme perdue et d'éliminer l'entrée en cache lui correspondant. Ceci nécessite l'ajustement d'un nouveau paramètre du système en fonction de l'environnement d'exécution (peut-être hétérogène) du système. De plus, le coût lié à l'enregistrement dans les messages de cette information n'est pas très important, en comparaison au coût fixe par paquet envoyé sur un réseau IP. Nous opterons donc pour la première solution.

Pour modéliser cette solution, nous considérons que les messages contiennent une "pile" reprenant les références des noeuds auxquels la requête a été transférée.

Lorsqu'un noeud n reçoit un message $request(K, Nodes)$, il le traite selon le pseudo-code 2.3. Lorsqu'un noeud n reçoit un message $answer(K, Responsible, Nodes)$, il suit le pseudo-code 2.4. Quand un noeud effectue une recherche pour une clé K donnée, il exécute la procédure décrite par le pseudo-code 2.5.

Un exemple de lookup récursif est proposé à la figure 2.2. La requête émise par $N62$ pour la clé 31 est propagée par $N14$ vers $N25$ car ce dernier est le résultat de l'opération de calcul du *nexthop* sur la clé 31 par $N14$. Lorsque $N25$ reçoit la requête, il s'aperçoit (de la même façon qu'en mode itératif) que son successeur $N32$ est le responsable de la clé 31, il émet une réponse contenant les références de $N32$ à destination de $N14$, qui

Pseudo-code 2.3 (Traitement d'une requête en mode récursif)

```
NextNode=n.Closest_preceding_node(K)
if (NextNode==n)
then
    Prec=pop(Nodes)
    Send(Prec,Answer(K,n.successor,Nodes))
else
    push(Nodes,n)
    Send(NextNode, request(K,Nodes))
end
```

Pseudo-code 2.4 (Traitement d'un message de réponse en mode récursif)

```
if (empty(Nodes)) then
    Show("Lookup for key "+K+" returned "+Responsible)
else
    Prec=pop(Nodes)
    Send(Prec,Answer(K,Responsible,Nodes))
end
```

Pseudo-code 2.5 (Recherche d'une clé en mode récursif)

```
n.lookup(Key K)

   NextNode=n.Closest_preceding_node(K)
   if (NextNode==n)
   then
       Show("Lookup for key "+K+" returned "+n.successor)
   else
      newStack(Nodes)
      push(Nodes,n)
      Send(NextNode, request(K,Nodes))
   end
```

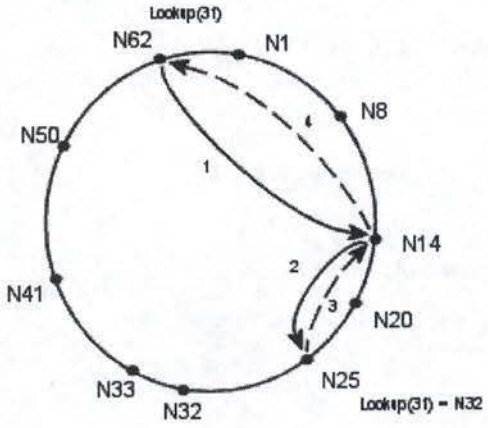

FIG. 2.2 – Exécution récursive d'un "lookup" sur la clé 31 à partir du noeud $N62$.

transfère cette réponse vers l'émetteur initial de la requête, $N62$.

2.1.3 Le mode récursif avec retour direct du résultat

Ce mode fonctionne selon le même principe que le mode récursif. La différence réside dans le fait que l'émetteur initial de la requête est spécifié dans les messages de recherche et ce, afin que le noeud qui trouve le résultat puisse le contacter directement pour le lui transmettre (il n'y a donc plus besoin de maintenir une pile des participants à la requête). Ce mode ne peut être décrit au moyen d'un pseudo-code reprenant des appels de procédure à distance et des instructions $return()$. En effet, la nuance entre ces deux modes réside dans la sémantique de cette instruction. Ce mode correspond plus précisément à un appel de méthode asynchrone, le noeud trouvant le résultat effectuant le "Call Back" sur l'émetteur initial de la requête. Les messages seront de deux types : $request(K, Source)$ et $answer(K, Node)$. Le champ $Source$ d'un message $request$ référence le noeud qui a initié la requête. Le champ $Node$ d'un message $answer$ référence le responsable effectif de la clé K.

Le pseudo-code 2.6 décrit la recherche d'une clé dans ce mode et le pseudo-code 2.7 le traitement d'une requête. La procédure $Wait\_Answer(K, Responsible)$ initialise $Responsible$ à la valeur enregistrée dans une réponse reçue avec K comme clé.

La figure 2.3 fournit une illustration de ce mécanisme. Nous remarquons bien que la

Pseudo-code 2.6 (Recherche d'une clé en mode récursif allégé)

```
n.lookup(Key K)

   NextNode=n.Closest_preceding_node(K)
   if (NextNode==n)
   then
      return(n.successor)
   else
      Send(NextNode, request(K,n))
      Wait_Answer(K,Responsible)
      return(Responsible)
```

Pseudo-code 2.7 (Traitement d'une requête en mode récursif allégé)

```
   NextNode=n.Closest_preceding_node(K)
   if (NextNode==n)
   then
      Send(Source,answer(K,succesor))
   else
      Send(NextNode, request(K,Source))
   end
```

seule différence par rapport au mode récursif "pûr" (voir figure 2.2) se situe lors du retour du résultat. Ici, $N25$ transmet directement sa réponse à $N62$.

2.2 Modes d'envoi d'un message

Le mode d'envoi d'un message peut également influencer le fonctionnement de Chord. Un noeud peut contacter un autre noeud en lui envoyant un message UDP ou en établissant systématiquement une connexion TCP/IP avec celui-ci. On peut également tenter de profiter d'une connexion déjà établie entre deux noeuds afin d'économiser, quand cela est possible, l'initialisation d'une connexion TCP/IP. Cette méthode peut également permettre à un noeud A de communiquer un message à un noeud B via une connexion $B \rightarrow A$ alors qu'une connexion ne peut être établie dans le sens $A \rightarrow B$. Cette solution semble donc réduire l'impact potentiel de l'invisibilité d'un noeud sur le système.

Hormis les facteurs de fiabilité, les deux premiers modes (UDP et établissement systématique de connexion TCP/IP) sont équivalents en terme d'influence de l'invisibilité d'un noeud sur le système.

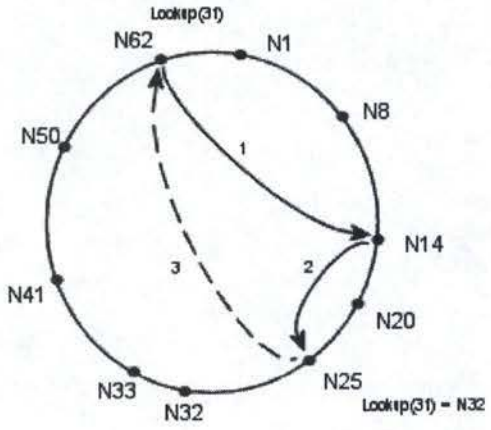

FIG. 2.3 – Exécution récursive d'un "lookup" sur la clé 31 à partir du noeud $N62$ avec retour direct du résultat.

2.3 Maintien de connexions TCP/IP

Si l'existence de connexions durables entre des noeuds peut avoir une influence sur le fonctionnement de Chord, la robustesse d'un système face aux imperfections de la couche réseau dépend forcément de la politique de maintien de connexions qu'appliquent les pairs qui le constituent.

On peut raisonnablement supposer qu'un noeud tente de maintenir des connexions TCP / IP avec son successeur ainsi qu'avec les autres entrées de sa "finger table". Rappelons que la finger table d'un noeud est constituée par les noeuds avec lesquels il vérifie la relation R_{node}. En effet, la présence d'un noeud A dans la "finger table" d'un noeud B implique que ce dernier, lors d'un lookup pour une clé K, contacte A si celui-ci est le résultat de l'application de l'opérateur *Closest_preceding_node* sur K. Le maintien d'une connexion avec le prédécesseur peut sans doute nuancer l'impact de l'invisibilité des noeuds sur le système. Nous examinerons précisément cet impact au chapitre suivant.

2.4 Conclusion

Une étude de l'impact de l'invisibilité d'un noeud doit être discutée en fonction des diverses variantes d'implémentation de Chord. En effet, ces variantes entraînent une uti-

lisation différente de la couche réseau inférieure par les noeuds et donc une influence différente de l'imperfection de cette couche sur le système.

Chapitre 3

Impact de l'invisibilité d'un noeud sur Chord

Dans ce chapitre, nous critiquons une hypothèse implicite concernant l'environnement de fonctionnement de Chord. Celle-ci porte sur l'accessibilité des noeuds par leurs pairs. Nous expliquons pourquoi, en pratique, cette hypothèse peut ne pas être respectée.

Ensuite, nous montrons en quoi les différentes variantes d'implémentation ne répondent plus au modèle de manière équivalente lorsque l'environnement de fonctionnement de Chord n'est pas idéal. Nous analysons l'impact de l'hétérogénéité de l'environnement, en fonction des catégories identifiées au chapitre précédent, sur la capacité du système à recevoir un nouveau noeud et sur le succès de l'exécution des primitives de stabilisation. Nous déduisons de cette analyse que, dans certaines situations, Chord peut se comporter de manière incohérente.

Enfin, au vu de cette analyse, nous identifions les caractéristiques d'implémentation qui rendent Chord le plus robuste possible vis-à-vis des imperfections de son environnement d'exécution. Ces caractéristiques définissent la politique de maintien de connexions, le mode d'envoi des messages et le mode de propagation des requêtes utilisés par les noeuds. Nous proposons également une adaptation simple de l'algorithme de routage de Chord et nous en analysons les faiblesses. Cette adaptation permet de limiter l'impact négatif de l'inaccessibilité des noeuds faisant partie d'un système Chord.

3.1 Une hypothèse forte sur la couche réseau inférieure

La définition actuelle de Chord semble poser des hypothèses implicites au sujet de la couche réseau sur laquelle les implémentations basées sur ce protocole sont susceptibles de fonctionner. Ainsi, selon la spécification des mécanismes de stablisation du réseau, un noeud doit pouvoir transférer des requêtes à un ensemble de noeuds. Ceci implique évidemment que ce noeud peut joindre chaque noeud de cet ensemble via la couche réseau inférieure. Or, le protocole Chord est voué à être mis en place sur des réseaux ne répondant pas obligatoirement à cette hypothèse puisque ses auteurs le destinent à être employé comme service de lookup pour des applications Internet ; et il est clair que les machines présentes sur le réseau Internet n'ont pas la certitude de pouvoir joindre n'importe quelle autre machine présente sur celui-ci.

En effet, l'utilisation d'un firewall [4] par un noeud ou un groupe de noeuds peut remettre en question l'accessibilité de celui-ci par ses pairs dans le système. De même, la présence d'un noeud accédant à l'Internet depuis un réseau local partageant une connexion grâce au mécanisme de "Network Adress Translator" [5] peut également rendre incertaine cette accessibilité. Plus simplement, des règles de routage défavorables peuvent empêcher un noeud d'accéder à une adresse IP donnée, bien que celle-ci soit effectivement utilisée par un noeud. Notons que cette dernière possibilité est souvent temporaire étant donné qu'elle résulte généralement d'une erreur de configuration de routeur. Il peut cependant arriver qu'un réseau d'entreprise se voit limiter ses accès aux réseaux extérieurs (sans les supprimer totalement) suite à la mise hors-service d'une connexion Internet commerciale et la subsistance d'une connexion privée vers d'autres réseaux de l'entreprise ou de ses partenaires.

Ces considérations mènent à penser qu'un noeud doit pouvoir joindre un système Chord et refuser, de par son inaccessibilité, de répondre à des requêtes émises par un de ses pairs.

Dans la suite de ce chapitre, nous qualifierons un noeud de "noeud caché" ou de "noeud invisible" lorsqu'il n'est pas accessible par l'ensemble de ses pairs. Ce concept de noeud caché couvre les exemples décrits ci-dessus.

Le fait qu'un noeud soit caché d'un autre n'implique pas forcément que ces deux noeuds ne pourront communiquer entre eux. En effet, si le noeud est caché de l'autre par un firewall, une connexion TCP / IP dans l'autre sens pourra sans doute être établie.

3.2 Impact sur le succès de l'insertion d'un noeud dans le système

En théorie, pour qu'un noeud puisse intégrer le système, il suffit qu'il soit capable de contacter un des participants de celui-ci. Cette possibilité de contact s'exprime par la connaissance de l'adresse d'un noeud du système et par la capacité effective du noeud à joindre cette adresse. Cette contrainte est propre à tous les protocoles Peer-to-Peer qui ne sont pas "hybrides".

En pratique, cependant, il faut absolument, en outre, qu'aucune "erreur critique" ne se produise durant l'exécution de la procédure d'insertion de celui-ci. Cette procédure correspond à un lookup transmis au "contact d'entrée" suivi de l'application des procédures de stabilisation relatives à l'enregistrement du résultat du lookup comme la référence vers le successeur du noeud entrant. Le traitement d'une requête produit une erreur dite critique lorsque cette erreur se reproduit à chaque ré-exécution du traitement tant que la disposition des noeuds dans le système reste inchangée. Plus précisément, l'environnement d'exécution d'une primitive qui produit une erreur critique ne peut être "amélioré" par les procédures de stabilisation de Chord de sorte que celle-ci ne produise plus d'échec.

Quand l'exécution d'une primitive d'accès au réseau produit un échec, le noeud ne peut évidemment pas rentrer dans le système. Si cette erreur est critique, le noeud est banni, de par la définition d'une erreur critique, tant qu'un changement favorable dans la disposition des noeuds du système ne s'est pas produit.

Les deux sections suivantes discutent de l'impact de l'insertion d'un noeud dans le système en fonction du mode de transfert de message.

3.2.1 Par établissement systématique de connexion

En mode itératif, une erreur critique est provoquée si un des noeuds qui participent à la requête ne peut contacter le noeud qui tente de joindre le système et inversément. Dans ce cas, le noeud est banni du système tant que l'environnement d'exécution de la requête ne change pas favorablement. Un exemple d'entrée de noeud Chord selon ce mode, illustrant la nécessité de la capacité des noeuds à établir certaines connexions est, donné à la figure 3.1. Les flèches représentent le déroulement de la primitive de *join* du noeud $N31$ dans le système ($N64$ est le noeud de contact) en mode itératif. Elles représentent également toutes les conditions de connectivité nécessaires pour qu'aucune erreur critique ne se produise.

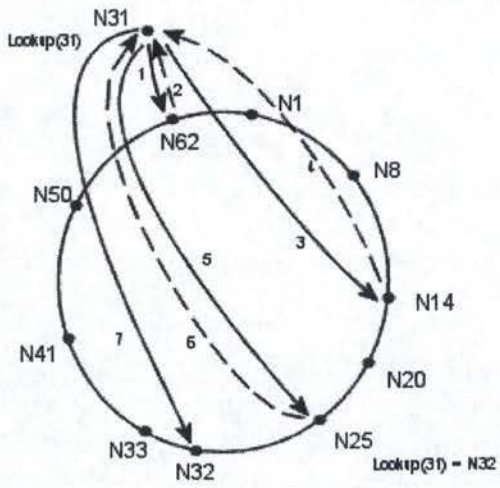

FIG. 3.1 – Introduction d'un noeud dans le système selon le mode de résolution itératif des requêtes.

En mode récursif, une erreur critique est provoquée si le contact d'entrée ne peut joindre le noeud entrant ou si l'environnement du lookup nécessaire à l'insertion du noeud est tel que la requête émise par le contact échoue[1] (voir figure 3.2).

Les flèches représentent le déroulement de la primitive de join du noeud $N31$ dans le système ($N64$ est le noeud de contact) en mode récursif. Elles représentent également toutes les conditions de connectivité nécessaires pour qu'aucune erreur critique ne se produise.

En mode récursif avec retour direct du résultat, le noeud est banni si le noeud qui trouve le résultat (en l'occurence le prédécesseur potentiel du résultat) ne peut le joindre. Le noeud est également banni si la requête ne parvient pas au prédécesseur potentiel du résultat pour des raisons d'invisibilité des noeuds intermédiaires.

3.2.2 Par utilisation de connexions pré-établies

En mode itératif, le noeud réussit à joindre le système lorsqu'il peut joindre tous les noeuds qui participent à la requête. Dans le cas contraire, il est banni du système jusqu'à ce que l'environnement de celui-ci change favorablement (i.e. jusqu'à ce que ce dernier permette la réussite de la requête). L'amélioration réside dans le fait que chaque noeud

[1] Les causes d'échec d'un lookup sont discutées à la section 3.3.

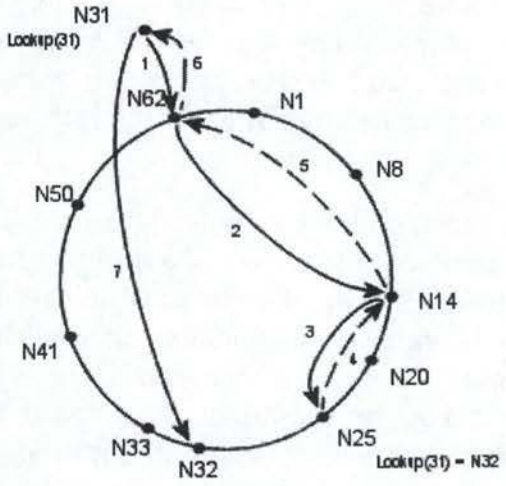

FIG. 3.2 – Introduction d'un noeud selon le mode récursif de résolution des requêtes.

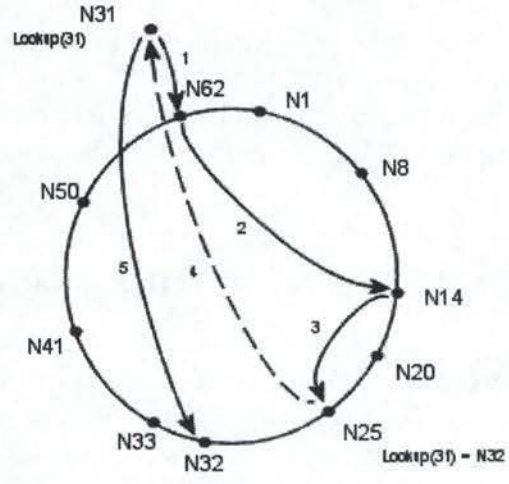

FIG. 3.3 – Introduction d'un noeud dans le système selon le mode de résolution des requêtes de type récursif avec retour direct du résultat.

peut répondre à la requête du noeud entrant par l'intermédiaire de la connexion que ce dernier a établie. Les flèches en trait discontinu de la figure 3.1 ne constituent donc plus des conditions de connectivité pour la réussite d'une entrée dans le réseau. Cette amélioration peut être rendue effective seulement si les connexions établies par le noeud entrant ne sont pas fermées avant l'envoi du résultat par les noeuds interrogés. Il faut donc veiller à ce que les noeuds ne ferment pas "trop tôt" les connexions qu'ils établissent pour émettre une requête.

En mode récursif, le noeud rentre dans le système à condition que l'environnement soit tel que la requête parvienne au prédécesseur du résultat. En effet, chaque noeud pourra utiliser, pour émettre sa réponse, la connexion utilisée par le noeud ayant retransmis la requête. Ceci réduit considérablement le nombre de conditions nécessaires à la réussite de l'exécution de la primitive d'entrée d'un noeud. Dans la figure 3.2, les flèches en trait discontinu ne constituent plus de conditions de connectivité pour la réussite du join. On remarque que, contrairement au mode itératif, les conditions de connectivité pour un noeud s'appliquent uniquement sur des noeuds présents dans sa table de routage. Donc, si les noeuds maintiennent des connexions avec les pairs présents dans leur table de routage, les résultats peuvent suivre ces connexions jusqu'à l'émetteur de la requête. Ceci implique l'économie d'établissements de connexion et tend donc à réduire le délai de transmission d'une requête à travers le système.

En mode récursif avec retour direct du résultat, le noeud est banni si le noeud qui trouve le résultat (en l'occurence son prédécesseur potentiel) ne peut le joindre. De nouveau, un échec peut également se produire si l'environnement d'exécution de la requête ne permet pas à tous les noeuds participant à celle-ci de joindre le noeud résultat de l'exécution de l'opérateur *Closest_preceding_node*. Etant donné que le chemin décrit par la requête et son résultat n'emprunte jamais deux fois la même connexion, l'utilisation possible d'une connexion pré-établie ne renforce pas le système.

3.3 Impact sur la stabilité du système

Si un noeud ne peut être accédé par un sous-ensemble de ses pairs, il est possible que certaines requêtes ne s'effectuent pas de manière correcte ou se terminent sans produire de résultat. Or, pour assurer la cohérence de Chord, tous les noeuds doivent communiquer avec un sous-ensemble de leurs pairs. En effet, la stabilité du système est assurée grâce à l'exécution distribuée de requêtes. Il s'avère donc opportun d'évaluer les risques d'échec de ces primitives de stabilisation lorsque l'environnement du système n'est pas idéal. Après avoir évalué ces risques d'échec, nous serons en mesure d'en évaluer l'impact sur la cohérence du système.

Dans la sous-section suivante, nous nous concentrons sur l'impact de l'invisibilité d'un noeud sur le comportement de son prédécesseur. Ensuite, nous observons l'impact de l'invisibilité des noeuds sur la réussite d'une requête et nous identifierons les caractéristiques d'implémentation d'un noeud Chord qui limitent au plus cet impact.

3.3.1 Comportement du prédécesseur d'un noeud caché

Le prédécesseur d'un noeud caché n'est pas forcément incapable de joindre directement ce noeud. Le noeud peut, en effet, correspondre à la définition de noeud caché parce qu'un noeud tiers ne sait pas établir de connexion avec lui.

Si le prédécesseur P d'un noeud N peut joindre ce même noeud N, alors il le conserve comme successeur jusqu'à son départ ou jusqu'à l'arrivée d'un noeud intermédiaire. Les primitives de demande de prédécesseur, de lookup et de lookup en vu de la mise à jour de la table de routage d'un noeud s'exécuteront sans erreur dans le cas d'une utilisation possible de connexions pré-établies et à condition que N puisse joindre P dans le cas de connexions systématiques.

Si P ne peut joindre N, alors un phénomène d'oscillation se produit dans le cas de connexions systématiques. En effet, quand P tente de joindre N et que sa tentative échoue, il considère N comme "mort" et exécute son mécanisme de résolution de nouveau successeur. Celui-ci aura pour résultat le successeur de N, S_N. Sachant que tout lookup pour une clé comprise entre P et N aboutit à P, P décidera que le résultat de ce lookup est son successeur actuel S_N. N étant présent dans le système, il détient la responsabilité des clés comprises entre P et N. Le système répond donc de manière incohérente à un lookup. Ceci peut avoir un impact plus néfaste qu'un échec simple, étant donné que le noeud émettant la requête tentera d'obtenir la ressource souhaitée en contactant S_N. Ce faisant, le noeud qui cherche la ressource se rend compte de l'erreur alors qu'un ensemble de procédures de rapatriement ou d'appropriation de ressources ont déjà été exécutées.

Lorsque P exécute son mécanisme de stabilisation, il prend à nouveau connaissance de l'existence de N. Comme celui-ci est mieux placé que S_N, il considère N comme son nouveau successeur. Les mécanismes de stabilisation prévoient que, lors de l'obtention de nouveau successeur, le noeud se fasse connaître de celui-ci afin qu'il mette à jour son prédécesseur. En l'occurence, P doit envoyer un message de notification à N. Comme cet envoi échoue par hypothèse, il exécute directement un "recovery" pour trouver un nouveau successeur. On remarque donc que P oscille perpétuellement d'un état de successeur invalide (S_N, dans le cas présent) à un état de successeur valide mais non joignable (à savoir, N). Ce deuxième état n'est conservé que très peu de temps.

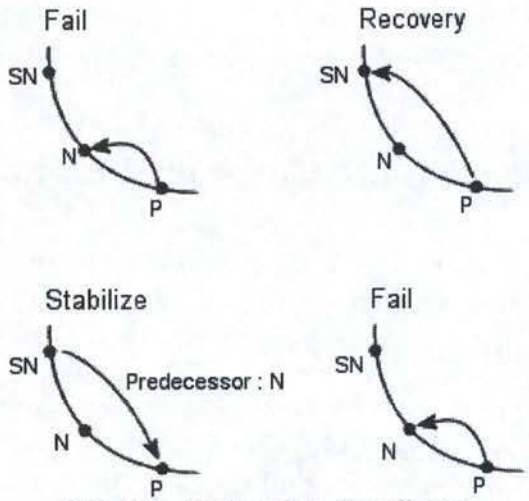

FIG. 3.4 – Phénomène d'oscillation.

On peut donc conclure qu'en général, et selon les conditions d'exécution décrites plus haut, une requête pour une clé K comprise entre les identifiants de P et de N produit un résultat incohérent sauf pendant la période (très courte) correspondant à la seconde partie de l'oscillation.

Dans le cas d'une utilisation possible de connexions pré-établies, la politique de maintien des connexions TCP/IP dispose d'une influence indubitable sur le comportement de P. Si cette politique impose à tout noeud du système de maintenir une connexion avec son prédécesseur, alors si N peut joindre P, cette connexion est établie et P peut l'utiliser pour émettre des requêtes et conserver une vue correcte de son successeur.

Il faut remarquer que cette connexion n'est établie que si N est mis au courant de l'existence de son nouveau prédécesseur. Comme P ne peut joindre N, il faut qu'un noeud tiers le prévienne de l'existence de P. Ceci peut être fait lors de l'exécution des mécanismes de stabilisation. Lorsqu'un noeud contacte son successeur, le successeur met à jour son prédécesseur si celui-ci est mieux placé. Il peut également fournir au noeud qui l'a contacté une liste de prédécesseurs établie de manière similaire à la "successor list". Grâce à cette liste, le noeud peut connaître un ensemble de prédécesseurs possibles sans attendre qu'un d'entre-eux le contacte. De cette façon, la connaissance du prédécesseur grâce à un noeud tiers est établie et une connexion peut être effectuée entre le noeud et son prédécesseur.

Il existe un mécanisme plus économe, qui nécessite l'envoi de cette liste de prédécesseur uniquement quand l'instabilité d'une partie du système est détectée par un noeud.

En effet, lorsqu'un noeud N reçoit des messages de notification de plusieurs noeuds, on peut déduire que, à part son prédécesseur réel P, les noeuds qui envoient ces messages ont une "idée" incorrecte de l'identité de leur successeur. Si nous considérons que cette situation est sans doute due à un phénomène d'oscillation causé par l'invisibilité de P, il est souhaitable pour N d'avertir son prédécesseur P que des noeuds considèrent falacieusement N comme leur successeur, alors que P est probablement le véritable successeur d'un de ces noeuds. Donc, lorsque N répond à une notification de P, il attache la liste des noeuds "en erreur" à sa réponse. Sur base de cette liste, P décide de l'identité de son véritable prédécesseur. Peut-être a-t-il, lui aussi, une vue incorrecte de son prédécesseur uniquement parce qu'il n'était pas au courant de son existence et que son véritable prédécesseur ne peut le joindre. Dans ce cas, P tente d'établir une connexion avec ce nouveau prédécesseur qui quitte, dès ce moment, son état oscillatoire. En effet, il peut utiliser cette connexion "en sens inverse" pour communiquer avec son successeur.

Si N ne peut joindre P ou si la politique de connexion n'impose pas aux noeuds du système de maintenir une connexion avec leur prédécesseur, le phénomène d'oscillation décrit plus haut se produit. Bien entendu, ce phénomène d'oscillation se produit également si N n'est pas au courant de la présence de P. En effet, dans ce cas, N n'établit pas de connexion avec P et P ne peut pas utiliser celle-ci comme parade à l'invisibilité de N.

En général, l'inaptitude d'un noeud à joindre son successeur aura pour effet de bloquer toute requête passant par lui, étant donné que le successeur d'un noeud est responsable de la construction de la table de routage de celui-ci. Ce blocage ne se produit pas quand le noeud prend un autre successeur pour palier la "mort" de son successeur effectif. En effet, ce nouveau successeur peut construire sa table de routage. Cependant, cette dernière ne respecte pas la spécification du système, celle-ci prévoyant que, à terme, tous les noeuds disposent d'une vue correcte de leur successeur et d'une table de routage construite à l'aide de requêtes envoyées à celui-ci.

3.3.2 Comportement général du système

Toute requête reçue par un noeud pour une clé K produit un échec si ce noeud doit transférer la requête et qu'il ne peut communiquer avec le noeud résultat de l'opération *Closest_preceding_node* appliquée à K. Si un noeud ne peut joindre directement un responsable de ressource trouvé par lookup, alors la ressource ne peut être accédée par le noeud. Si un noeud N ne peut pas communiquer avec son successeur, alors il bloque l'accès aux ressources dont les clés sont comprises entre son identifiant et l'identifiant de son successeur, et ce, en vertu du phénomène d'oscillation décrit plus haut.

Pour rendre Chord le plus robuste possible sans changer la spécification du modèle,

il faut sélectionner les variantes d'implémentation qui maximisent la probabilité qu'un noeud puisse communiquer avec son successeur. De manière plus générale, il faut réduire la probabilité de survenance d'une erreur critique lors de l'exécution d'une requête.

On peut donc conclure que, parmis les différentes variantes d'implémentations de Chord, la combinaison des trois facteurs suivants rend Chord plus robuste vis-à-vis de l'imperfection de la couche réseau sur lequel il fonctionne.

1. Utilisation possible de connexions pré-établies

2. Mode récursif de propagation des requêtes

3. Maintien de connexions avec l'ensemble des noeuds de la finger table et avec le prédécesseur

Il faut remarquer que le mode récursif n'est pas plus robuste que le mode itératif en terme de connectivité. En général, le nombre de conditions nécessaires à la réussite d'une requête est indépendant du choix d'un de ces deux modes. L'avantage du mode récursif réside dans le fait qu'un noeud ne contacte que des noeuds présents dans sa finger table. Ceci évite à un noeud de devoir effectuer un nombre important de connexions pour effectuer une requête. En effet, chaque noeud peut profiter de connexions pré-établies avec les noeuds de sa finger table pour transférer des requêtes. Cette différence constitue un avantage non négligeable si on prend en compte le délai dû à l'établissement de l'ensemble des connexions nécessaires à l'exécution de la requête.

Le maintien d'une connexion avec le prédécesseur n'est utile que s'il est accompagné par le mécanisme de distribution de listes de prédécesseurs (ou de sa version économe) explicité ci-dessus.

3.4 Amélioration de l'opérateur CLOSEST_PRECEDING_NODE

Pour réduire le nombre de survenances d'erreurs critiques, un noeud peut tenter d'affiner le calcul du *Closest_preceding_node* correspondant à une clé K afin de trouver le noeud le plus proche qu'il peut joindre, et pas le noeud le plus proche dans l'absolu. Si tous les noeuds procèdent de cette manière et que chacun d'entre-eux trouve un noeud répondant, à la fois, aux critères de localité et aux critères d'accessibilité, alors la requête produit un résultat. Dans ces conditions, le critère minimal assurant la cohérence de Chord est la capacité de chaque noeud à communiquer avec son successeur. En effet, une requête n'est pas propagée par un noeud quand son successeur est responsable de la clé recherchée (voir la figure 1.4). Donc, si il y a propagation, la clé recherchée est située entre le successeur et le noeud dans l'anneau. Dans ce cas, le successeur répond au critère de

localité et d'accessibilité. Une requête peut donc se propager de successeur en successeur jusqu'au prédécesseur du responsable de la clé. On peut donc ajouter la caractéristique suivante aux trois caractéristiques proposées dans la section précédente.

4. Utilisation d'un opérateur *Closest_preceding_node* prenant en compte les critères d'accessibilité des noeuds d'une finger table.

Si ce critère est suffisant pour la cohérence du système, il ne l'est pas du point de vue de l'efficacité des requêtes. En effet, si les noeuds ne transfèrent leurs requêtes qu'à leur successeur, la complexité d'un lookup est $O(N)$, N étant le nombre de noeuds dans le système. Cette complexité est trop importante pour pouvoir assurer la "scalabilité" du système.

Cette solution garantit donc la cohérence du système (sous la condition de communication possible, pour chaque noeud, avec son successeur) au prix d'une perte possible d'efficacité.

3.5 Conclusion

Dans ce chapitre, nous avons identifié les caractéristiques d'implémentation qui ont tendance à augmenter la robustesse de Chord. Cependant, ces améliorations ne sont toujours pas suffisantes pour assurer la robustesse du protocole. Il peut donc s'avérer utile de proposer un nouveau modèle, fonctionnant selon les mêmes bases, qui garantisse la robustesse du système tout en minimisant les coûts supplémentaires en terme de trafic engendré. Pour ce faire, nous proposons d'établir un modèle qui n'impose pas la possibilité de communication directe entre tous les noeuds présents dans un anneau Chord. C'est le but du chapitre suivant.

Chapitre 4

Un mécanisme de proxy pour le contact d'un noeud caché

Dans ce chapitre, nous proposons un mécanisme rendant Chord robuste vis-à-vis des imperfections de la couche réseau au-dessus de laquelle il repose. Ce mécanisme lève le critère minimal d'accessibilité évoqué au chapitre précédent. En effet, cette extension de Chord permet de conserver la cohérence du système même, si aucune connexion n'est possible entre un noeud et son successeur. Il permet également à un noeud de transférer une requête vers un noeud présent dans sa table de routage, même si il ne peut le joindre de façon directe.

La solution proposée ne change le fonctionnement de Chord que lorsque l'environnement du système est tel que l'utilisation des primitives classiques de Chord placerait le système dans un état incohérent.

Nous présentons une brève description du principe de fonctionnement de notre solution, illustrée par un exemple. Ensuite, nous explicitons les mécanismes nécessaires à la conservation de la cohérence du protocole. Nous fournissons également le pseudo-code correspondant à un *lookup* et permettant le support des proxy's. Enfin, nous énonçons une série d'optimisations possibles, et nous en évaluons l'impact potentiel sur les performances du système.

4.1 Idée générale

Le système se base sur le principe que chaque noeud maintient, tant que faire se peut, une connexion TCP/IP avec son successeur et avec tous les noeuds qui constituent des

entrées dans sa table de routage ou "finger table". Quand, pour des raisons de règles de routage ou de firewall, un noeud ne sait pas accéder directement à un de ces noeuds, il cherche à établir un chemin de noeuds ayant des connexions TCP/IP entre eux, débutant par lui même et finissant par le noeud qu'il désire joindre.

4.2 Principe de fonctionnement

Partant du principe que chaque noeud fonctionne de la sorte, et qu'une requête est toujours "forwardée" par un noeud vers un pair constituant une entrée dans sa finger table, on peut affirmer que le chemin emprunté par une requête est un chemin décrivant une suite de noeuds ayant une connexion TCP/IP directe entre-eux.

Les requêtes s'exécutent de manière récursive. Un noeud émet une requête vers un de ses proches et attend simplement une réponse de celui-ci.

Lorsqu'un lookup a pour but d'établir un successeur ou une entrée dans la table de routage, ce lookup doit être paramétré de façon à ce que son résultat soit constitué par le noeud cible et par la liste des noeuds qui ont participé au déroulement de la requête. Le noeud cible étant le noeud responsable de la clé qui a servi de paramètre au lookup.

Lorsque le noeud initiateur du lookup reçoit le résultat, il tente d'établir un chemin de connexions vers le noeud cible en commençant par le chemin direct. En cas d'échec, il tente d'établir une connexion avec le noeud suivant dans la liste et ainsi de suite jusqu'à ce qu'une connexion soit établie. On peut remarquer qu'une connexion existe entre le noeud inititateur et le dernier élément de la liste et que donc, sauf départ d'un des éléments de celle-ci, une connexion sera toujours trouvée.

Un "join" fonctionnera selon le même principe.

Par exemple, admettons un système Chord simple contenant deux noeuds. Les identifiants de ces noeuds sont 0 et 5. Leurs adresses sur le réseau sont respectivement : 1 et 6. Les deux tableaux suivants représentent les tables de routage des deux noeuds de ce système.

node(id=0 ip=1)		
Entrée	Clé	Route vers le Nexthop
1	1	node(id : 0 ip : 1) node(id : 5 ip : 6)
2	2	node(id : 0 ip : 1) node(id : 5 ip : 6)
3	4	node(id : 0 ip : 1) node(id : 5 ip : 6)
4	8	idle(Loopy entry)
5	16	idle(Loopy entry)
6	32	idle(Loopy entry)
Pred	–	node(id : 0 ip : 1) node(id : 5 ip : 6)
Connexions sortantes	–	node(id : 5 ip : 6)

node(id=5 ip=6)		
Entrée	Clé	Route vers le Nexthop
1	6	node(id : 5 ip : 6) node(id : 0 ip : 1)
2	7	node(id : 5 ip : 6) node(id : 0 ip : 1)
3	9	node(id : 5 ip : 6) node(id : 0 ip : 1)
4	13	node(id : 5 ip : 6) node(id : 0 ip : 1)
5	21	node(id : 5 ip : 6) node(id : 0 ip : 1)
6	37	node(id : 5 ip : 6) node(id : 0 ip : 1)
Pred	–	node(id : 5 ip : 6) node(id : 0 ip : 1)
Connexions sortantes	–	–

Si un noeud d'adresse 10 et d'identifiant 10 rentre dans le système, il aura pour successeur le noeud $(id : 0, ip : 1)$. Si son contact d'entrée est $(id : 5, ip : 6)$, celui-ci lui répondra à la requête par le chemin vers son successeur, à savoir

$$[node(id : 5, ip : 6), node(id : 0, ip : 1)].$$

Le noeud 10 peut alors tenter d'établir une connexion avec le noeud 0. Si la connexion ne se fait pas, il tente d'établir une connexion avec le noeud 5 car ce noeud est le suivant sur le chemin parcouru par la requête. Si cette dernière connexion réussit, alors la table de routage de 10 se présente de la façon décrite ci-dessous.

node(id=10 ip=10)		
Entrée	Clé	Route vers le Nexthop
1	11	node(id : 10 ip : 10) node(id : 5 ip : 6) node(id : 0 ip : 1)
2	12	node(id : 10 ip : 10) node(id : 5 ip : 6) node(id : 0 ip : 1)
3	14	node(id : 10 ip : 10) node(id : 5 ip : 6) node(id : 0 ip : 1)
4	18	node(id : 10 ip : 10) node(id : 5 ip : 6) node(id : 0 ip : 1)
5	26	node(id : 10 ip : 10) node(id : 5 ip : 6) node(id : 0 ip : 1)
6	42	node(id : 10 ip : 10) node(id : 5 ip : 6) node(id : 0 ip : 1)
Pred	–	node(id : 10 ip : 10) node(id : 5 ip : 6)
Connexions sortantes	–	node(nodeId : 5 nodeIp : 6)

Lorsque le noeud 10 devra envoyer un message au noeud 0, il l'enverra à 5 qui retransmettra à 0. Le chemin que doit parcourir le message est inscrit dans celui-ci et mis à jour lors de chaque saut.

4.3 Conservation de la cohérence du protocole

Pour conserver la cohérence du protocole, il faut que, lorsque survient un évènement de type départ ou arrivée d'un noeud voisin ou échec d'un noeud ou encore tout autre évènement susceptible d'avoir pour conséquence la modification du successeur d'un noeud ou d'une entrée dans sa table de routage, ce noeud se comporte de façon à faire tendre le système vers un environnement respectueux des principes évoqués plus haut.

Tel qu'il est décrit, le protocole n'impose pas le maintien d'une connexion directe entre un noeud et son successeur. Il n'impose pas non plus que cette connexion soit possible. La détection de la mort du successeur, facteur crucial de cohérence pour Chord, ne peut donc plus reposer sur la réussite ou non d'une connexion entre un noeud et son successeur. Ceci serait absurde vis à vis de l'idée générale de l'utilisation de proxys.

Une solution envisageable pour éviter ce problème réside dans l'instauration d'une borne de temps de réponse du successeur. Etant donné que, pour stabiliser le système, chaque noeud demande le prédécesseur de son successeur, le dépassement d'un temps limite de réponse à cette requête peut être considéré comme un échec du successeur. Le noeud ne recevant pas de réponse peut ainsi appliquer les mécanismes permettant la résolution d'un nouveau successeur.

Si dans l'exemple précédent, le noeud 0 disparaît du système. Le noeud 10, ne recevant plus de réponse de 0, exécute son mécanisme de recovery et trouve 5 comme nouveau successeur.

L'inconvénient de cette méthode réside dans l'introduction d'une contrainte dont la valeur doit être ajustée en fonction de l'environnement d'application du système. Si le temps limite de réponse est long, alors un noeud mettra en général plus de temps pour se rendre compte de l'échec de son successeur. Par contre, si la valeur de cette limite est petite, alors un noeud risque de considérer falacieusement son successeur comme mort. Le temps entre l'envoi de la requête et la réception de la réponse est peut-être long parce que le réseau sur lequel passent la requête et la réponse à celle-ci est lent. Le nombre de pairs intermédiaires peut aussi constituer un élément d'accroissement du temps de réponse à une requête.

Pour assurer la survie d'un système Chord, il faut imposer à chaque noeud de connaître

ses s successeurs consécutifs. La valeur de s doit être déterminée sur base de la propension des noeuds à quitter brutalement le système. Lorsqu'un noeud quitte volontairement le système, il peut prévenir son prédécesseur de son départ et lui fournir les références nécessaires à la résolution d'un nouveau successeur. Lorsque le noeud qui quitte le serveur ne prévient pas son prédécesseur, ce denier doit effectuer un "recovery", sur base de la liste des successeurs qu'il détient, pour établir son nouveau successeur. Si nous voulons assurer la survie du système en considérant les aspects d'invisibilité des noeuds, nous devons redéfinir la sémantique de s. En effet, s ne doit plus seulement définir le nombre de successeurs consécutifs qu'un noeud N doit connaître, mais bien le nombre de chemins de connexions actives vers les successeurs consécutifs de N, ne passant pas par le successeur $Succ_N$ de N. En effet, étant donné que ces routes servent lors du départ du $Succ_N$, il semble raisonnable d'imposer que les routes permettant de joindre les nouveaux successeurs potentiels de N ne nécessitent pas de communication avec $Succ_N$. Le mécanisme proposé est la recherche incrémentale des s successeurs de N tels que la route la plus courte calculée par N ne contienne pas $Succ_N$. En théorie, cette recherche peut ne jamais se terminer. Il faut donc imposer aux noeuds d'arrêter la recherche de tels successeurs lorsque le dernier successeur obtenu est à une distance de N supérieure à une valeur imposée $MaxRange$.

4.4 Pseudo-code des primitives Chord

Dans cette section, nous présentons les algorithmes de résolution de clé et de résolution de nouvelle entrée dans une table de routage correspondant à notre extension du protocole.

4.4.1 Résolution de clé

Un booléen, *path*, est enregistré au sein d'une requête. Il définit le type de réponse. Si *path* vaut "true" alors le résultat sera le chemin de la requête vers le responsable de la clé id. Si *path* vaut "false" alors le résultat sera uniquement le noeud responsable de la clé id.

Un résultat "tracé" est donc la suite des noeuds participant à la requête avec, entre-eux, la suite des noeuds qui servent de passerelle entre ceux-ci. L'opération *traceroute* appliquée sur un noeud x retourne la route la plus courte enregistrée vers ce noeud x. Notons que cette route existe toujours étant donné que les noeuds sur lesquel cette opération est appliquée sont toujours des noeuds figurant dans la table de routage du noeud qui effectue l'opération.

Pseudo-code 4.1 (Résolution d'une clé selon l'extension proposée)

```
n.find_successor(id,path) {
   if id in (n,n.successor())
   {
      if path
         return ((n.successor()).traceroute()).append([(n.id,n.ip])
      else
         return [(n.successor().id,n.successor().ip)]
   }
   else
   {
       n2 = closest_preceding_node(id)
       if path
       {
          pathn2 = n2.traceroute()
          return ((n2.find_successor(id,true)).append(pathn2).append([(n.id,n.ip)])
       }
       else
          return n2.find_successor(id,false)
   }
}
```

Nous proposons le pseudo-code 4.1 pour la résolution d'une clé selon notre extension à Chord.

4.4.2 Construction de la table de routage

Cette primitive fonctionne de la même manière que celle décrite dans la définition de Chord. Seule la nature du résultat diffère, en ce sens qu'il contient, en plus du responsable de la clé correspondant à l'entrée, le chemin parcouru par la requête pour aboutir à ce responsable. Le noeud qui recevra la réponse décidera du chemin de connexions le plus court et l'enregistrera dans sa table de routage.

L'instruction

```
finger[i]=n2.find_successor(n+exp(2,i-1))
```

est suffisante pour effectuer l'établissement la $i^{ème}$ entrée de la table de routage d'un noeud, selon la définition de Chord fournie dans [9]. Elle correspond à l'envoi d'une requête par un noeud d'identifiant n, destinée à son successeur $n2$, pour la clé de valeur $n + 2^{i-1}$.

Selon notre extension, l'identité seule du responsable de la clé de valeur $n + 2^{i-1}$ n'est

plus suffisante. Il faut également établir la route la plus courte vers celui-ci. L'algorithme ci-dessous décrit le fonctionnement de cette procédure modifiée.

```
nodepath = n2.find_successor(n+exp(2,i-1),true)
entry = find_a_way(nodepath)
finger[i]=entry
```

La procédure $find\_a\_way(nodepath)$ consiste à parcourir la liste de noeuds *nodepath* afin d'établir une connexion TCP/IP avec un des noeuds qui la constituent. Ceci doit être fait en commençant par la cible et puis, en cas d'échec, par les noeuds les plus proches de celles-ci sur le chemin.

4.5 Voies d'optimisation

Si la solution proposée est suffisante pour assurer la correction de Chord dans un environnement non idéal, elle n'en reste pas moins perfectible sur certains aspects. Nous proposons ici quelques améliorations susceptibles de réduire le coût (en terme d'utilisation du réseau) induit par notre solution.

4.5.1 Calcul distribué de la route la plus courte

Nous pouvons remarquer que la construction de la route la plus courte (à partir du chemin emprunté par une requête) repose, dans notre système, uniquement sur les possibilités d'accès du noeud qui a effectué la requête. Si tous les noeuds qui refusent des connexions de leurs pairs refusent les connexions de tous leurs pairs, cette méthode convient parfaitement, étant donné que tous les noeuds peuvent découvrir, sur base des tentatives de connexion qu'ils effectuent, les noeuds qui refusent les connexions et calculer la meilleure route.

Si, au contraire, certains noeuds sont accessibles uniquement par un sous-ensemble des membres du réseau, le noeud qui tente de calculer la route la plus courte ne dispose pas, sur base du chemin emprunté par une requête et de l'information qu'il peut obtenir en effectuant des tentatives de connexion, de toute l'information nécessaire au calcul de la route la plus courte.

Supposons que le chemin emprunté par une requête est $[N, A_1, A_2, A_3, C_1, C_2, C_3]$, N étant l'initiateur de la requête. Admettons que les noeuds A_i sont accessibles par N et que les noeuds C_i sont inaccessibles par N. Si N effectue le calcul de la route la plus courte de la façon décrite dans notre solution, il obtiendra la route $[N, A_3, C_1, C_2, C_3]$.

Or, rien ne dit que C_1 ne peut joindre directement C_3. Il se peut en effet que la requête soit passée par C_2 tout simplement parce que C_2 était le résultat du calcul (en C_1 ou en amont) du *nexthop* pour la clé recherchée. Si C_1 peut joindre C_3, alors il existe une route plus courte que celle calculée par $N : [N, A_3, C_1, C_3]$.

Dans un réseau Chord tel que des sous-ensembles de ses membres sont cachés des autres mais dont les membres de chacun de ses sous-ensembles peuvent se connecter l'un à l'autre, il peut être plus performant d'effectuer le calcul de la route la plus courte de manière distribuée, en permettant à chacun des membres d'une route enregistrée dans une table de routage d'un noeud de proposer un "raccourci" à ce noeud.

Nous pouvons appliquer ce principe en définissant un nouveau type de message (ou en introduisant un "drapeau" spécial dans un type de message pré-existant) envoyé par chaque noeud lorsqu'il vient de recevoir une réponse à une requête lui permettant d'établir une entrée de sa table de routage. Ce message sera envoyé sur la route la plus courte calculée localement par l'initiateur de la requête. Ce calcul de la route la plus courte sera ré-effectué lors de chaque saut du message. Le noeud qui reçoit un tel message tente de raccourcir cette route en appliquant le mécanisme de calcul de la route la plus courte sur le reste de la route à parcourir par le message. Lorsque le message parvient à la destination finale de la route, il est renvoyé dans sa version "élaguée" à l'initiateur qui peut remplacer la route calculée localement par celle-ci. Notons que ce mécanisme peut également être appliqué lors du retour de la requête initiale, en ajoutant un champ aux messages de réponses des requêtes visant à construire une entrée dans la table de routage d'un noeud.

4.5.2 Trace des lookups

Si la couche réseau au dessus de laquelle fonctionne un système Chord est parfaite, c'est à dire si tous les hôtes de celui-ci peuvent joindre leurs pairs, il est évident que le système de proxy engendrera un surcoût. On peut arguer que le système n'envoie un plus grand nombre de messages que lorsque ceux-ci sont nécessaires au fonctionnement du système. En effet, si une passerelle doit être établie, on peut affirmer que le système se trouve dans un environnement tel que, dans une implémentation "insouciante" de l'état du réseau, un noeud se trouve dans une situation où il doit établir un contact avec un autre noeud et qu'aucun mécanisme ne lui permet de le faire. Il n'empêche que le sytème de trace augmente la taille des messages envoyés. Si le système est parfait ou presque parfait, il y a une probabilité forte que ce surcoût soit inutile. Il pourrait donc s'avérer intéressant d'envisager une implémentation de noeud où une requête serait effectuée d'abord sans demande de trace et ré-émise avec demande de trace si la première ne donne pas un résultat permettant de mettre les noeuds en contact.

Avant de considérer ce genre d'optimisation, il pourrait d'abord s'avérer utile d'analyser la structure des messages émis lors du fonctionnement d'un système Chord. Si on constate que, en situation réelle, les paquets envoyés, correspondant à l'envoi de messages, ont une charge utile négligeable par rapport à leur coût fixe (somme des tailles des entêtes de trames ou de paquet), il vaudra peut-être mieux laisser les mécanismes effectuant d'office une requête avec demande de trace.

4.5.3 Recherche du meilleur chemin de connexion

Si un noeud A ne peut établir de connexion TCP/IP avec un noeud B, il n'est pas dit que l'inverse soit vrai également. De façon plus générale, un lookup de A vers B engendre une trace totalement indépendante de la trace qui serait produite lors d'un lookup de B vers A. Lorsqu'un noeud accepte une connexion, il pourrait donc tenter d'évaluer la longueur (le nombre de sauts ou la somme des délais entre les noeuds) du chemin TCP/IP et essayer d'établir une connexion avec le noeud initiateur du chemin. Les deux noeuds concernés pourraient décider de conserver celui qui a la longueur la plus courte.

Ce mécanisme entraîne un coût supplémentaire en terme d'utilisation du réseau. Il s'avère donc nécessaire d'évaluer l'opportunité d'un tel mécanisme.

Dans un système relativement stable, où les noeuds ne voient pas régulièrement de changements dans leur table de routage, un chemin établi entre deux noeuds est susceptible de rester valide pendant une longue periode de temps. Pendant cette période, un nombre important de messages de stabilisation peut passer par ce chemin. Le coût lié à la recherche d'un chemin dans l'autre sens peut donc paraître négligeable au vu du gain qu'il rapporte, pourvu que le nouveau chemin soit plus court.

Si l'environnement du système Chord est instable, c'est à dire si des arrivées et des départs de noeuds se produisent fréquemment, alors les tables de routage des noeuds changent régulièrement. Les chemins enregistrés dans celles-ci ne sont pas valables pour une longue période de temps. Il est donc peut-être trop coûteux de chercher à établir de meilleurs chemins entre deux noeuds alors que ceux-ci ne resteront sans doute plus liés, dans un futur proche, par la relation R_{node}[1].

[1] Cette relation implique la connaissance de deux noeuds et donc l'enregistrement de l'un dans la table de routage de l'autre.

4.5.4 Utilisation du mode de propagation récursif avec retour direct du résultat

Si, lors du transfert d'une requête par un noeud, celui-ci enregistre dans le message le chemin déjà parcouru par la requête, alors le noeud qui trouve le résultat peut chercher le noeud le plus proche de l'émetteur de la requête (parmis la suite des noeuds enregistrés dans le message) qu'il peut joindre et envoyer le résultat à celui-ci afin que le message ne parcoure pas l'entièreté du chemin emprunté par la requête. Dans le pire des cas, ce mécanisme sera équivalent au mode de propagation récursif et assurera donc un chemin fiable vers l'émetteur initial de la requête.

Encore une fois, le coût lié aux tentatives d'établissement de connexion avec les noeuds intermédiaires peut sembler trop important par rapport au gain fourni par la minimisation de la longueur du chemin.

4.5.5 Détection de routes invalides

Lorsqu'un noeud transmet une requête sur une route enregistrée dans sa finger table, il ne peut s'assurer de la bonne réception du message par le destinataire. Etant donné que ces routes sont reconstruites périodiquement, un noeud ne maintiendra jamais éternellement de route invalide dans sa table. Toutefois, le temps de réaction d'un noeud disposant d'une route invalide peut constituer un facteur trop important d'échec de requête. Un mécanisme simple de message d'erreur peut donc s'avérer utile pour éviter un tel problème.

Chaque noeud connaît le chemin parcouru par une requête reçue. Lorsqu'il se trouve dans une situation où il doit retransmettre la requête et qu'il ne peut joindre le noeud auquel il est censé la retransmettre, il envoie un message d'erreur sur le chemin inverse au chemin déjà parcouru par la requête concernée. L'émetteur initial de la requête est ainsi prévenu de l'échec de celle-ci et peut la transmettre de nouveau jusqu'à ce qu'un chemin valide vers le destinataire final existe.

4.5.6 Détection de boucles dans les routes

Il est possible qu'un noeud N participe plus d'une fois à la résolution d'une clé dans le système. Il peut y participer un nombre indéterminé de fois en tant que proxy, et une seule fois en tant que *nexthop* calculé par un noeud pour cette requête. La route reçue par le noeud initiateur de la requête contiendra donc plusieurs fois le noeud N. Il faut donc que le calcul local de la route la plus courte tienne compte de ces boucles.

Notons que le calcul distribué de la route la plus courte supprime automatiquement ces boucles.

4.5.7 Utilisation de l'opérateur Closest_preceding_node amélioré

Dans la définition actuelle du protocole, chaque noeud enregistre, pour chaque entrée dans sa table de routage, une liste de noeuds qui représente le chemin à parcourir par une requête lorsqu'elle est transférée vers cette entrée. Dans un environnement parfait, correspondant à l'hypothèse implicite de [9] où tous les noeuds peuvent joindre les autres noeuds du système, la taille des traces est de l'ordre de $O(Log(N))$ (N étant le nombre de noeuds présents dans le système), et les chemins effectivement enregistrés dans la table de routage de chaque noeud sont de longueur 2. Ceci est dû au fait que les noeuds enregistrent la route la plus courte vers leurs "fingers". Cependant, dans un environnement hétérogène, la longueur des routes enregistrées varie significativement en fonction des possibilités de communication entre les noeuds. Cette longueur ne peut donc être exprimée au seul moyen d'un rapport au nombre de noeuds présents dans le système. Théoriquement, la longueur de ces routes est de l'ordre de $O(N)$. Mais cette complexité correspond à un environnement improbable où chaque noeud peut joindre un et un seul noeud du système et où le graphe représentant ces possibilités de connexion est un circuit hamiltonien.

Dans un environnement "mitigé", où quelques noeuds seulement sont inaccessibles, il n'est peut-être pas nécessaire de mettre en oeuvre le principe de calcul de route par proxy pour toutes les entrées de la table de routage. Nous pouvons décider, par exemple, d'assurer ce principe pour le calcul de la route vers le successeur et d'utiliser l'opérateur *Closest_preceding_node* amélioré de telle façon que l'élection du *nexthop* pour une requête soit faite en considérant les critères d'accessibilité des noeuds enregistrés dans la table de routage.

4.5.8 Amélioration de la politique de connexion

Dans des situations extrêmes, où la répartition des noeuds sur l'anneau n'est pas homogène, un ensemble important de noeuds peut avoir, au sein de sa table de routage, une référence vers un même noeud. Au vu de la politique de connexion proposée, ce genre de situation risque de mener un noeud à accepter un nombre de connexions important. Si la politique de maintien de ces connexions a été proposée afin d'augmenter la probabilité de possibilité de communication entre deux noeuds du système, celle-ci peut-être revue de manière à ce que les noeuds ne l'appliquent que lorsqu'elle contribue effectivement à

l'augmentation de cette probabilité.

En pratique, cela se traduit par le maintien sur une longue durée d'une connexion entre deux noeuds seulement dans le cas où une connexion dans l'autre sens est impossible et que les noeuds participant à la connexion sont régulièrement amenés à se transférer des messages. Concrètement, ce genre de situation se produit lorsqu'un noeud vérifie la relation R_{node} avec un autre noeud et qu'une connexion ne peut être établie dans un des deux sens. De la même façon, cette situation se produit pour un noeud servant de proxy entre deux noeuds se trouvant dans la situation précitée.

4.6 Conclusion

Dans ce chapitre, nous avons proposé une extension aux mécanismes de Chord afin de permettre l'utilisation de ce protocole sur des réseaux tels que certains noeuds ne peuvent communiquer (directement) entre eux pour les raisons énoncées au chapitre 3. Nous avons identifié les choix d'implémentation nécessaires au bon fonctionnement de notre extension. Ainsi, nous avons défini comment les noeuds doivent découvrir la sortie du réseau de leur pairs ainsi que le mode de propagation des requêtes.

Nous avons présenté les adaptations des algorithmes de résolution de clé ainsi que les adaptations au niveau de la méthode de constitution des listes de successeurs pour rendre ce mécanisme conforme à notre "philosophie" d'accessibilité des noeuds.

Enfin, nous avons énoncé une série de principes nouveaux visant à augmenter les performances de notre solution.

Maintenant, il nous faut évaluer la correction de notre extension et tenter de quantifier la perte de performance induite par notre solution. Il faut également montrer que l'extension ne diminue pas les performances de Chord lorsque l'environnement est idéal. Nous allons tenter d'illustrer ces propos en analysant des simulations de notre système et en les comparant avec les résultats de simulations du système Chord classique placé dans un environnement non idéal.

Chapitre 5

Description des simulateurs

Dans ce chapitre, nous proposons une solution permettant de simuler un système Chord dans un environnement réseau non idéal.

Pour ce faire, nous décrivons l'interface d'un module qui permet de simuler la transmission de messages entre les membres d'un réseau dont la topologie ne permet pas à chaque membre de se connecter à tous ses pairs. Cette description correspond à la première section du chapitre.

Ensuite, nous décrivons trois simulateurs de systèmes Chord. Ces descriptions constitueront les sections suivantes du chapitre.

Le premier simulateur fonctionne selon les règles décrites dans [9], tandis que le deuxième fonctionne avec le support des proxy's. Le troisième et dernier simulateur n'utilise le système de proxy que pour l'envoi de messages entre un noeud et son successeur.

Pour décrire ces simulateurs, nous spécifions l'ensemble des variables d'état d'un noeud, le comportement d'un noeud lors de la réception d'un message et l'ensemble des procédures de stabilisation que les noeuds Chord exécutent périodiquement.

Etant donné que le fonctionnement d'un noeud Chord classique et le fonctionnement d'un noeud Chord correspondant à la variante allégée de notre solution peuvent être spécifiés comme des cas particuliers du fonctionnement d'un noeud Chord supportant les proxy's, nous commençons par définir le simulateur correspondant à cette dernière solution.

5.1 Le réseau

Cette section décrit un module dont les primitives correspondent aux interactions d'un hôte avec le réseau auquel il est connecté. Ce module dispose de primitives supplémentaires permettant de spécifier si le réseau simulé permet à une machine d'adresse IP donnée d'établir une connexion vers une autre adresse IP donnée.

Dans la sous-section qui suit, nous expliquons en quoi ce module est nécessaire à la réalisation de simulations portant sur notre problématique. Ensuite, nous décrivons l'interface fournie par ce module.

5.1.1 Intérêt de la simulation de la couche réseau

Une perception simplifiée du fonctionnement d'un réseau suffit pour mettre en évidence l'impact de l'inaccessibilité d'un noeud sur un système Chord. Le but recherché dans l'émulation du réseau au sein du simulateur est de permettre la spécification d'une "matrice d'accessibilité" représentative du réseau. Cette matrice sert à représenter les caractéristiques pertinentes du lien entre chaque couple de machines sur le réseau. Elle sert essentiellement à déterminer la possibilité ou non d'établir une connexion TCP/IP entre chaque paire de noeuds.

Etant donné le caractère orienté de l'établissement d'une connexion, une demi-matrice ne peut suffire pour représenter l'état du réseau de manière suffisament précise. En effet, la possibilité pour un noeud d'établir une connexion vers un autre noeud n'implique pas que ce dernier puisse établir une connexion avec le premier. Deux machines, dont une seule est équipée d'un firewall interdisant l'établissement d'une connexion depuis l'extérieur, peuvent tout de même communiquer si la machine protégée établit elle même la connexion vers l'autre machine.

D'autres caractéristiques peuvent être enregistrées afin d'étudier une gamme différente de propriétés des systèmes Chord. Par exemple, l'enregistrement d'une valeur représentative du délai de transfert d'un message entre deux machines ou du délai d'établissement d'une connexion peut être établi afin d'introduire ces contraintes et d'analyser leur impact sur la performance du système.

5.1.2 Primitives de la couche réseau

Nous décrivons, dans cette section, les différentes primitives offertes par la couche réseau. Ces primitives sont accessibles par les noeuds du système. Dans certains cas, pour assurer la cohérence du simulateur et éviter de permettre aux noeuds d'effectuer des opérations qu'ils ne pourraient pas effectuer en réalité, nous spécifions des contraintes sur l'utilisation de ces primitives.

1. Insert(Node Ip)

Cette primitive enregistre le noeud <Node> à l'adresse IP <Ip> du réseau. Tout message transmis sur le réseau vers l'IP <IP> sera donc analysé par l'instance de noeud <Node>. Bien entendu, à chaque instant, un seul noeud peut être enregistré à une adresse IP donnée.

2. Delete(Ip)

Cette primitive supprime le noeud d'adresse IP <Ip> du réseau. Ceci correspond à la sortie d'un noeud du réseau.

3. SetConnectionPossible(Ip1 Ip2 Possible)

Cette primitive enregistre la possibilité ou non (selon la valeur booléenne de <Possible>), pour une machine dont l'interface réseau porte l'adresse IP <Ip1>, d'établir une connexion vers la machine d' adresse IP <Ip2>. Toutes les causes qui empêchent la réalisation d'une connexion sont donc couvertes par la simulation.

Il faut remarquer le caractère orienté de cette primitive. En effet, une machine A refusant, via son firewall, une connexion initiée depuis une machine B peut très bien établir une connexion avec la machine B si celle-ci est joignable par le réseau et qu'elle n'a pas de firewall configuré pour refuser une connexion initiée par A.

Cette primitive n'est évidemment pas accessible par les noeuds, étant donné qu'elle vise à définir la topologie du réseau.

4. IsConnectionPossible(Ip1 Ip2)

Cette primitive retourne une valeur booléenne correspondant à la possibilité d'établissement de connexion depuis l'hôte d'adresse IP <Ip1> vers l'hôte d'adresse IP <Ip2>.

En pratique, les noeuds du réseau n'ont pas accès à l'entièreté de l'information fournie par cette primitive. En effet, un hôte ne peut vérifier, par l'entremise du réseau auquel il est connecté, que la possibilité d'établir une connexion depuis lui-même et non depuis un

hôte quelconque. Il faut donc s'assurer que, pour tous les appels à cette primitive par un noeud d'Ip X, le paramètre <Ip1> vaut X.

5. IsConnected(Ip1 Ip2)

Cette primtive retourne une valeur booléenne représentant le fait qu'une connexion, établie par la machine d'IP <Ip1>, existe entre celle-ci et la machine enregistrée à l'IP <Ip2>.

Encore une fois, il faut assurer la cohérence du simulateur en certifiant qu'aucune machine tierce n'utilise cette primitive. En effet, un hôte ne peut collecter cette information sur un réseau que si il joue un des deux rôles dans la connexion potentielle.

6. Connect(Ip1 Ip2)

Cette primitive tente de réaliser une connexion entre les hôtes enregistrés aux IP <Ip1> et <Ip2>, avec l'hôte d'IP <Ip1> initiateur de la connexion.

Cette primitive se déroule avec succès si deux machines sont effectivement enregistrées à ces adresses, qu'une connexion dans ce sens soit possible, et qu'une connexion ne soit pas déjà établie dans ce sens. Cette dernière condition n'est pas propre au fonctionnement d'un réseau mais est ajoutée afin de ne pas multiplier les connexions entre deux hôtes.

Dans la pratique, il faut que tout noeud d'adresse Ip X utilise cette primitive avec une valeur du paramètre <Ip1> égale à X.

7. Disconnect(Ip1 Ip2)

Cette primitive coupe une connexion établie entre l'hôte d'IP <Ip1> et l'hôte d'IP <Ip2> (dans le sens <Ip1> vers <Ip2>). Elle réussit toujours et est sans effet si aucune connexion n'était établie dans ce sens.

A nouveau, seuls les noeuds jouant un des deux rôles dans la connexion peuvent appeler cette primitive.

8. SendMessage(Ip1 Ip2 Message)

Cette primitive tente de faire parvenir un message émis par l'hôte enregistré à l'adresse <Ip1> vers l'hôte enregistré à l'adresse <Ip2>. Les conditions de réussite de cet envoi sont : l'enregistrement d'un hôte à chaque adresse IP concernée et l'existence d'au moins une connexion établie entre-elles.

Un hôte ne peut appeler cette primitive qu'en affectant son adresse IP comme paramètre <Ip1>.

5.2 Le système avec proxy's

Ce simulateur correspond à la solution proposée au chapitre 4. Cette solution permet à un noeud, devant émettre de façon régulière des messages vers un noeud quelconque, de faire face à l'impossibilité d'établir une connexion avec ce dernier. Pour ce faire, les mécanismes de stabilisation du système Chord sont adaptés afin que, lorsqu'un noeud émet une requête sur le système dans le but d'établir ou de mettre à jour les entrées de sa table de routage, il reçoive l'information nécessaire (une suite de noeuds) à la communication, via un ou plusieurs noeuds tiers si nécessaire, avec le noeud résultat de sa requête.
Le système est écrit selon la variante récursive d'implémentation des requêtes. Ainsi, toute réponse à une requête "remonte" exactement le chemin emprunté par la requête pour arriver à son émetteur initial.

Nous décrivons le simulateur d'un système Chord avec proxy en spécifiant les différents messages envoyés par les noeuds, le comportement d'un noeud lorsqu'il reçoit un certain type de message et l'ensemble des procédures d'envoi périodique de messages de stabilisation.

5.2.1 Caractéristiques d'un noeud

Nous spécifions ici les différents attributs et variables d'état constitutives d'un noeud.

1. Id

Identifiant numérique du noeud calculé par hachage de son adresse IP.

2. Ip

Adresse Ip du noeud.

3. NetworkSize

Nombre maximal de noeud sur le réseau Chord.

4. FingerTable

Table de routage du noeud. Cette table contient l'ensemble des hôtes auxquels le noeud transfère les requêtes qu'il reçoit et dont il ne peut fournir la réponse. Pour la réalisation du mécanisme de proxy, cette table contient également les chemins permettant de joindre ces différents hôtes. La taille m de cette table satisfait l'égalité

$$m = \log_2 NetworkSize.$$

Comme nous l'avons déjà expliqué dans l'introduction sur Chord, tout noeud du système tente de maintenir cette table de manière à ce que tout pair enregistré à la $i$ème position dans la table soit le responsable de la clé dont la valeur est :

$$Id + 2^{i-1} \mod NetworkSize.$$

5. SuccessorList

Liste des successeurs consécutifs du noeud. Cette liste est nécessaire pour la résolution d'un nouveau successeur lorsque le successeur du noeud sort du réseau. La taille de cette liste doit être calculée en fonction de la valeur de $NetworkSize$ et de la propension des noeuds à sortir de façon simultanée du réseau Chord. Plus le système est instable au niveau des départs et arrivées de noeud, plus cette liste doit être grande afin de diminuer la probabilité qu'un noeud voie l'ensemble de ses successeurs sortir du réseau et perdre ainsi tout lien vers un successeur potentiel, ceci mettant en péril la cohérence du système.

6. PredList

Ensemble des noeuds ayant envoyé récemment un message de notification au noeud. Autrement dit, cette liste représente l'ensemble des noeuds ayant considéré récemment le noeud comme successeur.

7. PredTable

Table des prédécesseurs. Les implémentations actuelles du simulateur n'utilisent qu'une des références de cette table. Seul le prédécesseur direct du noeud est utilisé. Il est cependant envisageable qu'une implémentation d'un mécanisme supplémentaire nécessite qu'un noeud enregistre l'ensemble de ses prédécesseurs consécutifs. Cette table pourrait aider un noeud à augmenter sa connaissance des noeuds qui le précèdent, afin d'accroître la probabilité qu'un noeud établisse lui même une connexion vers un nouveau prédécesseur lorsque le prédécesseur courant quitte le réseau.

8. Connections

Registre des connexions sortantes établies (et toujours actives) par le noeud.

9. Timers

Un noeud exécute certaines actions de façon périodique et change son état lorsque certains types d'évenement ne se sont pas produits pendant une durée donnée. Il faut

donc que le noeud enregistre des informations concernant l'instant de survenance d'évènements et les fréquences d'exécution des procédures périodiques. Ces informations sont enregistrées dans les variables d'état décrites ci-dessous.

- PredUpdate_Time : temps de la dernière réception d'un message de notification en provenance du prédécesseur
- PredUpdate_Rate : délai toléré entre deux réceptions de ce type de message ; passé ce delai, le prédécesseur est considéré comme sorti du réseau.
- RequestPred_Time : instant du dernier envoi de la demande du prédécesseur au successeur.
- RequestPredDelay : délai d'attente de réponse du successeur ; passé ce delai, le successeur est considéré comme sorti du réseau.
- ConnectionDelay : délai toléré entre deux envois de message via une connexion donnée ; passé ce delai, la connexion est fermée.
- StabilizeDelay : période d'envoi de message de stabilisation au successeur.
- NotifyDelay : période d'envoi d'un message de notification au successeur.
- BuildFingersDelay : période d'envoi de requête de mise à jour des noeuds de la table de routage.
- KeepAliveDelay : période d'émission de messages visant à maintenir les connexions vers les entrées de la table de routage.

Nous avons, dans cette section, décrit toutes les variables d'état nécessaires au fonctionnement de Chord modifié pour le support de notre mécanisme de proxy's. Nous devons maintenant spécifier les traitements effectués sur les messages reçus par un noeud Chord, ainsi que les procédures exécutées périodiquement.

5.2.2 Comportement d'un noeud

Nous spécifions, dans cette partie, le contenu de chaque type de message et les traitements effectués par les noeuds sur les messages qu'ils reçoivent.

Lorsqu'un noeud est actif mais ne dispose d'aucun lien vers un noeud tiers (parce qu'il est le premier à rentrer dans le réseau Chord ou parce que le réseau s'est effondré à la suite de départs simultanés trop nombreux), la réception d'un message, quel qu'il soit, entraîne une (ré)initialisation des procédures périodiques et l'enregistrement de l'émetteur du message comme successeur du noeud récepteur. Afin de ne pas alourdir les spécifications des messages et des traitements, nous ne reprennons pas ce cas particulier dans chaque descriptif de message, même si celui-ci est considéré lors de chacun des traitements de message effectués par les noeuds.

Le type d'un message est reconnaissable grâce à l'analyse du champ type repris dans tous les messages. Nous discutons donc les traitements effectués par les noeuds en fonction de la valeur de ce champ.

Forward

Ce message n'est pas prévu dans la définition du protocole Chord reprise dans [9]. En effet, ce type de message a été introduit pour permettre le routage de messages à travers le réseau. Lorsqu'un noeud doit effectuer un envoi de message à un noeud en passant par au moins un noeud tiers, il encapsule le message à envoyer dans un message de type forward qui contient le message original et la route (reprise sous la forme d'une liste de couple (Identifiant de noeud, Adresse IP)) à parcourir par celui-ci.

Type : **forward**	
Champ	Descriptif
source	Emetteur initial du message
path	Chemin à suivre pour le message
message	Message original à envoyer au destinataire

Lorsqu'un noeud reçoit un message de ce type, le champ *path* contient une liste d'au moins deux noeuds. Le premier élément de la liste est le noeud qui reçoit ce message et le dernier élément de la liste est le noeud destinataire final du message.

Un message de ce type est traité de la manière suivante : si la longueur du chemin repris dans le champ *path* du message vaut 2, alors le noeud suivant,*Nexthop*, est le destinataire final du message. Le récepteur extrait le message repris dans le champ *message* et l'envoie,tel quel, au destinataire final dont l'adresse IP est enregistrée dans le deuxième et dernier élément de la liste *path*. Si la longueur du champ *path* est supérieure à 2, alors le *Nexthop* est un noeud proxy vers le destinataire final du message. Le noeud transfère le message reçu au *Nexthop* en supprimant la première entrée du champ *path* du message. L'adresse IP du noeud auquel il faut transférer le message est reprise dans le deuxième élément du champ *path* du message reçu (le premier après la mise à jour).

Par la suite, lorsque nous parlerons d'un envoi de message entre deux hôtes, nous ne préciserons plus si le message est encapsulé dans un message forward ou non. La règle pour un envoi est applicable pour tous les types de messages ; tout envoi de message sur un chemin de plus de deux hôtes (tout envoi de message via un ou plusieurs proxy's) est nécessairement un envoi de message de type forward.

Lookup

Ce message correspond à une requête visant à trouver le responsable d'une clé donnée. Ce message contient donc une valeur de clé constituant le critère de recherche. Il contient également le chemin qu'il a déjà parcouru. Ainsi, lorsqu'un noeud est en mesure de répondre à la requête reçue, il peut envoyer un message de réponse sur le chemin inverse à celui emprunté par la requête. Ce moyen d'appliquer le caractère récursif des requêtes n'est sans doute pas le plus économique en terme d'utilisation du réseau. En effet, les messages semblent gonflés d'informations alors qu'un mécanisme de mise en cache des requêtes aurait permis d'éviter cet alourdissement des messages.

Néanmoins, cette information de routage est nécessaire pour le récepteur d'une réponse. En effet, lorsqu'un noeud reçoit une réponse à une requête, il tente d'établir une suite de noeuds (la plus courte possible) ayant des connexions établies entre-eux et lui permettant de joindre le noeud spécifié dans la réponse à sa requête. Pour ce faire, il faut donc que le noeud qui reçoit la réponse soit mis au courant du parcours de sa requête. L'enregistrement du chemin parcouru au sein du message semble donc constituer une solution adéquate pour ce genre de traitements.

Un système de mise en cache des requêtes pour pouvoir assurer le retour des réponses selon une logique récursive ne permettrait pas au récepteur final de la réponse de connaître le chemin parcouru par sa requête. C'est la raison pour laquelle nous avons opté pour la première solution. Notons que le choix de ce type de solution a déjà été argumenté (sans considérer la nécéssité pour le noeud destinataire de connaître l'ensemble de la route parcourue) dans le chapitre 2 du mémoire.

Type : **lookup**	
Champ	Descriptif
key	Valeur de la clé dont l'émetteur recherche le responsable
path	Chemin déjà parcouru par la requête.

Lorsqu'un noeud reçoit ce type de message, il consulte sa table de routage pour voir si son successeur n'est pas le responsable de la clé spécifiée dans le champ *key* du message. Si son successeur est bien le responsable de la clé, le noeud construit un message de type *reply_lookup* en spécifiant le numéro d'identification et l'adresse IP de son successeur comme responsable de la clé. Il indique également le chemin maximal à parcourir (par

les messages envoyés par le destinataire de la réponse) pour joindre son successeur. Pour établir le chemin à parcourir, il ajoute au chemin parcouru par la requête (enregistré dans le champ *path* du message) le chemin qu'il utilise pour joindre son successeur. Finalement, il envoie la réponse à l'émetteur de la requête via le chemin inverse à celui parcouru par la requête reçue. Ce chemin est calculé en prenant l'inverse du champ *path* reçu dans le message initial.

Si son successeur n'est pas le responsable de la clé, il élit, parmis les noeuds repris dans sa table de routage, le *Nexthop* le plus approprié[1] pour répondre à cette requête. Il met à jour le champ *path* du message reçu en y ajoutant le chemin entre lui et le *Nexthop* élu. Ensuite, il envoie le message mis à jour au *Nexthop*.

finger_find_successor, join_find_successor

Ces messages sont traités de la même manière qu'un *Lookup*. Cependant, le type de message envoyé en réponse à l'émetteur d'une requête de ce type n'est pas *reply_lookup* mais bien *reply_finger_find_successor* ou *reply_join_find_successor*. Seul le type de message change par rapport à une réponse de type *reply_lookup*, le reste de la composition du message est identique.

Type : **finger_find_successor | join_find_successor**	
Champ	Descriptif
key	Valeur de la clé dont l'émetteur recherche le responsable
path	Chemin déjà parcouru par la requête.

L'intérêt de ces types de message réside dans le fait que l'émetteur d'une requête doit décider, au moment où il reçoit une réponse, quel traitement effectuer sur cette réponse. Les requêtes sont donc caractérisées par la raison pour laquelle elles ont été émises. Ces requêtes sont de trois types : les requêtes visant simplement à établir le responsable d'une ressource, les requêtes émises afin d'établir une entrée dans la table de routage d'un noeud et les requêtes visant à établir le successeur d'un noeud lorsqu'il rentre dans le système.

Ainsi, comme nous le verrons dans les descriptions qui suivent, un noeud recevant une réponse peut décider du traitement à effectuer sur base du seul type de réponse reçue. Il ne doit donc pas maintenir de variable d'état ou encore de cache des requêtes émises pour établir ce traitement.

reply_join_find_successor

Ce message constitue une réponse à une requête émise par un noeud entrant dans le système. Seul le noeud émetteur de la requête initiale recevra la réponse matérialisée par

[1]La manière dont cette élection se produit est expliquée dans le descriptif de Chord au chapitre 2.

un message de type *reply_join_find_successor*.

Type : **reply_join_find_successor**	
Champ	Descriptif
source	Emetteur de la réponse
path	Chemin parcouru par la réponse
key	Clé recherchée. Ici, correspond à l'identifiant numérique du noeud entrant
key_succ_id	Identifiant numérique du responsable de la clé key
key_succ_ip	Adresse IP du responsable de la clé key

Lorsqu'un noeud reçoit ce type de message, il enregistre comme successeur le noeud (*key_succ_id* , *key_succ_ip*). Il calcule le plus court chemin vers ce successeur sur base de la liste *path*, selon la méthode d'établissement de route expliquée au chapitre 4. Ensuite, il lance les procédures périodiques de stabilisation. Celles-ci sont décrites dans la sous-section 5.2.3.

reply_finger_find_successor

Ce message constitue une réponse à une requête émise par un noeud cherchant à mettre à jour sa table de routage. Seul le noeud émetteur de la requête initiale recevra la réponse matérialisée par un message de type *reply_finger_find_successor*.

Type : **reply_finger_find_successor**	
Champ	Descriptif
source	Emmetteur de la réponse
path	Chemin parcouru par la réponse
key	Clé recherchée. Ici, correspond au pas d'avancement recherché par le noeud
key_succ_id	Identifiant numérique du responsable de la clé key
key_succ_ip	Adresse IP du responsable de la clé key

Périodiquement, chaque noeud effectue une requête sur certaines clés remarquables (Id+1, Id+2, Id+4, Id+8, ...). Les réponses à ces requêtes constituent une base pour l'élaboration de la table de routage d'un noeud.

Lorsqu'un noeud reçoit ce type de message, il enregistre dans sa table de routage, au niveau correspondant à la clé remarquable *key*, les références du responsable de la clé (*key_succ_id* , *key_succ_ip*) ainsi que le chemin vers ce noeud.

request_predecessor

Ce message, servant à la stabilisation du système, est envoyé par chaque noeud, de manière régulière, à son successeur. Lorsqu'un noeud reçoit ce message, il consulte

l'enregistrement de son prédécesseur dans *PredTable* et construit une réponse reprenant les informations sur celui-ci.

Les noeuds enregistrent également leur liste de successeurs dans la réponse afin de profiter de l'envoi de celle-ci pour transmettre l'information nécessaire pour la résolution d'un nouveau successeur lors de l'échec de leur successeur courant.

Type : **request_predecessor**	
Champ	Descriptif
path	Chemin parcouru par la requête. Ici, ce chemin est égal au chemin qui lie le noeud émetteur de la requête et son successeur. Il correspond donc au chemin repris dans la première entrée de la finger table du noeud.
source	Identifiant et Adresse Ip du noeud émetteur de la requête

reply_predecessor

Bien entendu, ce message est reçu par un noeud en réponse à une requête de type *request_predecessor*. Il contient une référence vers le prédécesseur du noeud émetteur de la réponse ainsi que la liste des successeurs de ce même émetteur. Pour calculer la valeur du champ *path* enregistré dans la réponse, le noeud qui crée le message ajoute à la liste reprise dans le champ *path* du message *request_predecessor* reçu le chemin vers son prédécesseur enregistré dans *PredTable*.

Type : **reply_predecessor**	
Champ	Descriptif
source	Noeud émetteur de la réponse (source.id, source.ip)
path	Chemin maximal entre le récepteur de la réponse et le prédécesseur du noeud qui crée le message.
pred_id	Identifiant du prédécesseur du noeud émetteur de la réponse.
pred_ip	Adresse IP du prédécesseur du noeud émetteur de la réponse.
succ_list	Liste des successeurs du noeud émetteur de la réponse.

Lorsqu'un noeud reçoit un message de ce type, il consulte la champ *pred_id* pour vérifier s'il est bien le prédécesseur de son successeur. Si il est bien le prédécesseur de son successeur, il met à jour sa liste de successeurs sur base de la valeur du champ *succ_list*.

Si le noeud n'est pas le prédécesseur de son successeur et si *pred_id* est compris entre l'identifiant numérique du noeud et *source.id*, le prédécesseur reçu est un meilleur successeur pour le noeud courant ; celui-ci peut donc mettre à jour la première entrée de sa finger table et enregistrer un chemin vers celui-ci. Si *pred_id* n'est pas compris entre ces valeurs, alors le noeud émetteur de la réponse ne dispose pas du bon prédécesseur, car le noeud courant constitue un meilleur successeur pour le noeud émetteur. Cette

situation instable sera réglée lors de l'envoi d'un message *notify* du noeud courant vers son successeur.

notify

Ce message est envoyé périodiquement par un noeud vers son successeur.

Type : **notify**	
Champ	Descriptif
source	Noeud émetteur du message (source.id, source.ip)
path	Chemin parcouru par le message

Lorsqu'un noeud reçoit ce message, il enregistre dans *PredList* la source du message, le chemin vers celle-ci et l'heure à laquelle le message a été reçu. Ensuite, il compare l'identifiant de son prédécesseur avec la valeur de *source.id*. Si l'émetteur du message semble plus proche du noeud courant que son prédécesseur actuel, il met à jour l'entrée correspondante dans *PredTable*. Il envoie un message de réponse à l'émetteur du *notify* en incorporant au sein du message les valeurs enregistrées récemment dans *PredList*. Enfin, il met à jour la valeur de *PredUpdate_time*, censée refléter le dernier moment de la mise à jour du prédécesseur.

reply_notify

Lorsqu'un noeud met à jour la valeur de son prédécesseur suite à la réception d'un message de type notify, il envoie une réponse de type reply_notify contenant la valeur de sa *PredList*.

Type : **reply_notify**	
Champ	Descriptif
source	Noeud émetteur du message (source.id, source.ip)
path	Chemin parcouru par le message
predlist	Liste des noeuds ayant récemment envoyé un message notify à l'émetteur de ce message

Lorsqu'un noeud reçoit ce type de message, il parcourt l'ensemble des noeuds enregistrés dans le champ *predlist*. Considérant que ces noeuds ont comme successeur l'émetteur de ce message, il envoie un message en se proposant comme meilleur successeur à tout noeud de cette liste qui se situe entre lui et l'émetteur du message *reply_notify*.

better_succ

Ce message est envoyé par un noeud lorsqu'il pense qu'un noeud tiers devrait le considérer comme son successeur.

Type : **better_succ**	
Champ	Descriptif
source	Noeud émetteur du message (source.id, source.ip)
path	Chemin parcouru par le message

Lorsqu'un noeud reçoit ce type de message, il compare l'identifiant de son successeur avec *source.id*. Si *source.id* se situe bien entre le noeud récepteur du message et son successeur, alors il met à jour son successeur et la route vers celui-ci.

keep_alive

Ce message ne contient aucune autre information que son type. Il permet à un noeud de maintenir une connexion en vie en envoyant régulièrement ce type de message via cette connexion. En effet, les connexions sont gérées de telle façon qu'une fermeture de connexion n'est exécutée que lorsqu'aucun message n'a été transmis via celle-ci pendant un temps donné (*ConnectionDelay*).

5.2.3 Procédures de stabilisation

Tous les noeuds du système exécutent périodiquement un certain nombre de procédures visant à maintenir la cohérence de celui-ci et à mettre à jour les tables de routage ; cela afin de rendre le système le plus efficace possible et afin de lui permettre de s'adapter aux départs et aux arrivées qui surviennent en son sein.

Stabilize

Cette procédure exécutée selon une fréquence définie grâce à *StabilizeDelay* entraîne l'envoi périodique d'un message de type *request_predecessor* par les noeuds à leur successeur. Cette méthode permet de réagir à l'arrivée d'un noeud dans le système. En effet, son arrivée aura pour effet la mise à jour du prédécesseur par le successeur du noeud entrant. Ainsi, le prédécesseur de celui-ci sera averti, grâce à la réponse au message *request_predecessor*, de l'existence de ce nouveau noeud.

Notify

Cette procédure exécutée périodiquement, selon la fréquence définie par $NotifyDelay$, a pour but l'envoi régulier de messages de type $notify$ par les noeuds à leur successeur.

BuildFingers

Cette procédure, exécutée à la fréquence définie par $BuildFingersDelay$, est responsable de l'envoi périodique de requêtes de type $finger\_find\_successor$. Les réponses à

ces requêtes permettront aux noeuds émetteurs de celles-ci de mettre à jour les entrées enregistrées dans leur *FingerTable*. Grâce à ce mécanisme, la table de routage reflète, de façon quasi permanente, une vue partielle, mais correcte, du réseau par chaque noeud participant à celui-ci.

CheckPred

Cette procédure a pour but l'effacement de l'enregistrement du prédécesseur d'un noeud lorsque ce dernier n'a pas reçu de message *notify* de son prédécesseur pendant un long moment spécifié grâce à *PredUpdate_Rate*.

De la même manière, cette procédure enlève de la liste des prédécesseurs *PredList*, les noeuds n'ayant pas émis de message notify depuis un nombre de secondes plus grand que *NotifyDelay*.

CheckConnections

Cette procédure est exécutée par chaque noeud dans le but de fermer toute connexion sortante inutilisée. Cette procédure vérifie donc que le temps écoulé depuis le dernier passage de message sur toutes les connexions sortantes enregistrées dans *CurrentConnections* ne dépasse la valeur de *ConnectionDelay*. Le cas échéant, la connexion est fermée par le noeud qui a initialisé la connexion.

KeepAlive

Cette procédure a pour but de rendre les noeuds conformes à la politique de maintien de connexions appliquée dans le système. La politique annoncée pour cette solution visant le maintien d'une connexion (ou d'un chemin de connexions) par chaque noeud vers l'ensemble des entrées de sa table de routage et vers son prédécesseur, cette procédure est implémentée grâce à l'envoi périodique de message de type *keep_alive* sur les chemins repris dans la table de routage (*FingerTable*) et sur le chemin vers le prédécesseur (repris dans *PredTable*).

La réception ou l'envoi d'un message sur une connexion sortante d'un noeud ayant pour conséquence la mise à jour du temps de dernière utilisation d'une connexion dans *CurrentConnections*, les messages de types *keep_alive* permettent bien de maintenir ces connexions vivantes. Il faut évidemment veiller à ce que le delai entre ces envois de messages *keep_alive* (*KeepAliveDelay*) soit plus court que le temps maximal d'inactivité de connexion toléré (*ConnectionDelay*).

CheckSucc

Cette procédure a pour but de rendre les noeuds sensibles au départ de leur successeur.

Pour ce faire, un delai de réponse maximal (*RequestPredDelay*) aux requêtes de type *request_predecessor* est introduit. Cette procédure vérifie donc que le successeur d'un noeud réponde toujours à ces requêtes. Lorsqu'un successeur ne répond pas endéans le delai, il est considéré comme sorti du réseau et la procédure de résolution de nouveau successeur est exécutée.

La procédure de résolution d'un nouveau successeur se fait de la façon décrite au premier chapitre. Concrètement, quand un noeud N s'aperçoit du départ de son successeur, il consulte la liste de successeurs construite à partir des messages de stabilisation et cherche à établir un nouveau successeur parmis ceux de la liste de façon à élire le noeud dont l'identifiant est le plus proche de celui du noeud N.

5.3 Le système avec proxy's "allégé"

Cette variante du système de proxy a été réalisée dans le but d'apporter une solution robuste tout en limitant le nombre d'enregistrements de routes par les noeuds participant au réseau. Elle vise aussi à réduire le nombre de connexions établies par les noeuds du système. Cette solution constitue donc un compromis entre robustesse et efficacité. Notons qu'un système Chord reste correct s'il contient des noeuds des deux types. Nous pouvons dès lors imaginer une solution où les noeuds adapteraient leur comportement en fonction de la charge de leur table de routage ou en fonction du nombre de connexions qu'ils doivent maintenir. Lorsque les noeuds supportant le mode avec proxy voient leur tables surchargées, ils peuvent passer dans le mode "allégé" sans compromettre la correction du système.

Afin de réaliser ce compromis, le critère permettant d'affirmer que tout noeud peut joindre l'ensemble des noeuds de sa table de routage a été réduit à l'assurance d'accessibilité de tout noeud à son successeur. Ainsi, un noeud est assuré de pouvoir joindre son successeur alors que les routes vers les autres entrées de sa table de routage ne sont pas construites en profitant du système de proxy.

Tout comme le système "standard" de proxy, la version "allégée" fonctionne selon le mode récursif de propagation des requêtes. Cependant, le traitement des messages de type *reply_finger_find_successor* est effectué de façon différente. Ainsi, lorsqu'un message de ce type est reçu par un noeud N, l'informant que le pair responsable de la clé fournie dans le message *finger_find_successor* auquel il fait réponse est R, le noeud ne cherche plus à construire la route la plus courte possible vers R sur base du chemin parcouru par la réponse. En effet, N enregistre le chemin direct vers R.

Si le chemin enregistré de cette façon est tel qu'il est impossible pour le noeud N de transmettre un message vers le noeud R, c'est à dire si R n'est pas connecté à N et si une connexion initialisée depuis N vers R est impossible, alors la route est marquée comme inutilisable.

Selon cette solution, seul le chemin enregistré vers le successeur peut contenir des noeuds intermédiaires (proxy's).

Lorsqu'un noeud fonctionnant de la sorte doit transférer une requête, il consulte sa table de routage en faisant fi des routes inutilisables. Pour ce faire, une modification de l'opération *Closest_preceding_node* a été réalisée afin que celle-ci fonctionne selon l'optimisation proposée à la section 3.4. L'utilisation de cette méthode aura pour effet d'augmenter le nombre de sauts effectués par les messages. En effet, les requêtes auront tendance à être transférées vers des noeuds plus proches du responsable du transfert. Cependant, ces transferts ne seront effectués que de manière directe (sauf lors des transferts vers le successeur). En effet, l'utilisation de proxy est réservée à la communication entre les noeuds vérifiant la relation de (prédécesseur – successeur).

Si le critère d'évaluation des solutions est le nombre de sauts réels effectués pour l'aboutissement d'une requête, c'est à dire en comptabilisant l'utilisation d'un proxy comme un saut, la préférence pour l'une de ces deux solutions ne peut être justifiée sur une base entièrement théorique. En effet, la solution allégée a tendance à transférer les requêtes vers des noeuds plus proches de l'émetteur, mais ces transferts ne se réalisent que grâce à l'envoi d'un seul message sur une seule connexion. La première solution, quant à elle, privilégie le transfert d'une requête vers le noeud le plus proche du noeud susceptible d'y répondre. Cependant, ce transfert peut se réaliser par l'envoi d'un message de type *forward* sur un nombre indéterminé de connexions pour aboutir au noeud cible.

Le but des simulations est, entre autres, d'établir une comparaison entre ces deux solutions en terme de nombre moyen de sauts effectués par une requête.

Les caractéristiques du système allégé décrites ici constituent les seules différences entre les deux systèmes de proxy.

5.4 Le système sans proxy

Cette solution correspond à la définition du comportement d'un noeud Chord exposée dans [9]. Nous avons observé que, de par son mode de fonctionnement, on pouvait assimiler le fonctionnement d'un noeud Chord sans proxy au fonctionnement d'un noeud Chord avec proxy dont l'enregistrement des routes est restreint aux routes directes. On

peut donc facilement réaliser une implémentation d'un noeud Chord répondant à la description de [9] en effectuant des modifications simples sur le mode d'enregistrement des routes par tout noeud recevant de l'information de routage reçue via les messages de type *reply_join_find_successor*, *reply_finger_find_successor*, *reply_predecessor* et *reply_notify*.

Le mode de calcul de la plus courte route étant le même lors de tout enregistrement de route vers un noeud remarquable, tous ces calculs ont été regroupés au moyen d'une procédure nommée *ShortestPath*. Pour simuler le fait qu'un noeud ne considère jamais que les routes directes vers les noeuds présents dans sa table de routage, *ShortestPath* peut être modifiée afin de répondre toujours par la route directe.

Certains détails d'implémentation du fonctionnement d'un noeud disposant du support des proxy's ont été choisis pour permettre ce même support. Ainsi, le mode de détection d'échec du successeur se fait via l'utilisation d'un timer vérifiant le délai de réponse de ce successeur. Comme tous les liens sont directs en mode normal (sans proxy), cette méthode peut s'avérer trop complexe par rapport à la détection de l'échec uniquement sur base de la possibilité d'établissement de connexion avec le successeur.

Nous voulons quand même remarquer que ce genre de détail n'est pas explicité dans [9]. Il est par conséquent faux d'affirmer que nous modifions le fonctionnement de Chord en conservant la détection par timer de l'échec du successeur. En effet, [9] définit le comportement d'un noeud découvrant "l'échec de son successeur", mais ne définit pas comment un noeud décide de considérer son successeur comme étant "en échec". Aucune mention n'est faite au sujet de l'incapacité d'un noeud à joindre un de ses pairs alors que celui-ci participe véritablement au système. Ainsi, il n'existe pas de spécification du comportement d'un noeud recevant, par le biais d'un message contenant de l'information de routage, une référence vers un noeud actif qu'il ne peut atteindre.

5.5 Conclusion

Dans ce chapitre, nous avons présenté les différents modules de notre simulateur. Nous avons expliqué comment nous avons modélisé le réseau afin de pouvoir évaluer le fonctionnement de Chord dans un environnement où les noeuds n'ont qu'une accessibilité partielle vers l'ensemble de leurs pairs. Nous avons décrit en détail le fonctionnement d'un noeud de notre simulateur, et ce, selon la version initiale décrite dans [9] ainsi que selon nos deux variantes de Chord augmenté du support des proxy's.

Chapitre 6

Simulations

Afin d'illustrer nos propos théoriques, nous avons réalisé des simulations de systèmes Chord en faisant varier la topologie du réseau. Dans ce chapitre, nous décrivons le contexte des simulations, ou, plus précisément, les environnements dans lesquels nous avons placé des noeuds afin qu'ils interagissent pour créer un système Chord.

Ensuite, nous nous penchons sur les résultats obtenus lors des simulations afin d'évaluer les conséquences de l'inaccessibilité des pairs d'un système Chord tel qu'il est décrit dans [9]. Enfin, nous analysons les résultats des simulations portant sur les systèmes Chord augmentés du support des proxy's et de sa version allégée.

6.1 Contexte des simulations

Dans cette section, nous décrivons les environnements dans lesquels les performances de Chord et des extensions proposées sont évaluées.

Nous avons restreint les contraintes topologiques à la question de l'accessibilité ou non d'un noeud par l'ensemble de ses pairs. Ainsi, on peut considérer que tous les environnements générés sont des environnements formant une partition, à deux éléments, des noeuds du système. La première partie est constituée d'un certain nombre de noeuds accessibles par tous les noeuds du réseau. La partie résiduelle est formée par des noeuds qui refusent toute connexion externe.

Nous ne prendrons donc pas en considération les environnements tels que plusieurs noeuds sont inaccessibles par leurs pairs mais peuvent établir des connexions entre-eux. Ce qui est, par exemple, le cas des sous-réseaux protégés par un firewall et contenant

plusieurs participants au système.

La procédure employée pour la constitution des systèmes dans lesquels nous injectons les requêtes est telle qu'elle minimise l'impact (sur la constitution elle-même) du refus de connexion par certains pairs. Pour ce faire, nous introduisons d'abord les noeuds qui acceptent les connexions entrantes (nous dirons de ces noeuds qu'il font partie du "noyau"). Lorsque ces noeuds se sont stabilisés, nous introduisons les noeuds qui refusent les connexions entrantes (nous les qualifierons d'"externes"). Cette insertion se déroule rapidement par rapport à la fréquence de rafraîchissement des tables de routage des noeuds, étant donné que les insertions des noeuds externes sont exécutées de façon concurrente. De cette manière, les premiers noeuds externes qui rentrent dans le système ont peu de chance de participer (en raison de leur présence dans une table de routage) aux requêtes visant à insérer les derniers noeuds externes. Il n'y a donc presque aucune chance qu'un noeud externe soit exclu du système. Ceci permet d'assurer la rigueur des chiffres sur la cohérence des requêtes. En effet, si un noeud est exclu du système, toutes les requêtes pour les clés dont il est censé être responsable seront résolues de façon incohérente (par rapport au résultat attendu globalement). Nous excluons donc les simulations où un noeud externe est banni du système lors de son entrée.

Remarquons que cette attention ne doit être portée qu'au niveau des simulations concernant la version initiale de Chord. Bien entendu, aucune simulation de système Chord avec proxy n'a produit ce genre d'exclusion.

Les simulations portent toutes sur un système Chord dont l'espace des identifiants est de taille 2000 et contenant 100 noeuds. Lorsque les noeuds sont stabilisés, chaque noeud injecte des requêtes couvrant l'ensemble de l'espace des identifiants afin de "jouer" l'ensemble des scénarii possibles de résolution de requêtes initiées par lui. Chaque noeud d'identifiant n effectue une requête pour les clés valant

$$(n + 1 + (5 * x)) \ mod \ 2000 \text{ avec } x \text{ tel que } 1 \leq x < 400.$$

Ceci correspond à l'exécution, par chaque noeud, de 400 requêtes ; soit un nombre total de 40.000 requêtes par simulation. Notons que la méthode employée pour le calcul des clés à résoudre par chaque noeud est telle que, globalement, toutes les valeurs de clé sont couvertes avec une forte probabilité. Pratiquement, chaque noeud effectue une requête pour 400 clés parmis les 2.000 clés possibles, ces 400 clés étant uniformément réparties sur l'espace des identifiants.

6.2 Résultats

Dans un premier temps, nous simulons un système Chord "standard", en faisant varier progressivement le pourcentage de noeuds accessibles dans le système.

La tableau suivant reprend le pourcentage de requêtes ayant reçu une réponse cohérente, le pourcentage de requête ayant reçu une réponse incohérente et le nombre de sauts effectués par les requêtes résolues (c.à.d celles pour lesquelles l'émetteur a reçu une réponse), en fonction du pourcentage de noeuds inaccessibles. Nous présentons également le pourcentage de requêtes perdues, indicateur clé du manque de robustesse d'un système Chord déployé sur une topologie de réseau "hostile".

Pourcentage de noeuds inaccessibles	0	1	2	4	6	8	10	20
Réponses cohérentes (%)	100	97,40	94,19	87,69	81,82	76,53	72,67	49,72
Réponses incohérentes (%)	0	0	0	0,02	0,03	0,06	0,08	0,36
Requêtes perdues (%)	0	2,60	5,81	12,29	18,15	23,41	27,25	49,92
Nombre de sauts	5,11	5,11	5,11	5,10	5,09	5,09	5,08	4,95

Si le système fonctionne parfaitement lorsque tous les noeuds sont accessibles, nos simulations montrent que le pourcentage de requêtes résolues décroît fortement et linéairement avec le pourcentage de noeuds accessibles. Sachant que ces requêtes sont également employées pour stabiliser le système et assurer les performances en terme de nombre de sauts par *lookup* lorsque le système évolue, nous pouvons affirmer que l'inaccessibilité des pairs d'un système Chord remet en cause sa stabilité et les performances attendues pour celui-ci.

Comme ce type de requête est également employé pour l'entrée d'un noeud dans le système, le pourcentage de requête résolues reflète également l'espérance de succès d'une insertion de noeud dans le système. Par exemple, nous pouvons dire que, lorsqu'un système Chord respectueux de sa définition décrite dans [9] est déployé sur une topologie de réseau telle que 10 % des noeuds du systèmes sont inaccessibles, une requête visant à introduire un nouveau noeud dans ce système a plus d'une chance sur quatre d'être perdue et donc, d'empêcher le noeud de rentrer.

Notons que le nombre moyen de sauts effectués par une requête résolue diminue quand le pourcentage de noeuds inaccessibles augmente. Ceci est dû au fait que, la probabilité d'échec d'un saut de requête augmentant logiquement avec le nombre de noeuds inaccessibles, les requêtes parcourant un nombre moins important de sauts ont une probabilité plus forte de se terminer avec succès lorsque la probabilité d'échec par saut de requête augmente. Comme seules les requêtes résolues sont comptabilisées dans ce chiffre, nous

observons cette diminution de longueur moyenne.

Nous ne pourrons donc pas effectuer de comparaison de la solution "standard" de Chord avec notre solution, déclinée en fonction du nombre de noeuds inaccessibles dans le système. A défaut, nous comparerons la longueur moyenne des chemins parcourus par les requêtes selon notre solution avec la longueur moyenne des chemins parcourus par les requêtes dans un système Chord "standard" placé dans un environnement idéal.

Etudions maintenant la cohérence des réponses. Pour cet aspect, c'est la probabilité d'adjacence (sur l'anneau représentatif du réseau) de deux noeuds "externes" qui compte. Plus précisément, cette probabilité correspond à la probabilité que deux noeuds externes vérifient, entre eux, la relation de successeur/prédécesseur. Nous remarquons que cette probabilité s'avère assez faible (pour un pourcentage raisonnable de noeuds inaccessibles). Ainsi, peu de simulations ont révélé une incohérence dans les réponses aux requêtes. Les réponses incohérentes ont toujours été fournies par un petit nombre de noeuds du système. Ces noeuds "incohérents" sont des noeuds inaccessibles dont le successeur refusait également les demandes de connexions qui lui étaient adressées. Une illustration de ce genre de phénomène a été présentée à la figure 3.4.

Même si la valeur de ce pourcentage de réponses incohérentes est faible, elle révèle l'instabilité de Chord lorsqu'un nombre assez important de participants au système refusent les connexions entrantes. Cette instabilité se caractérise par le phénomène d'oscillation "perpétuelle" des noeuds concernés, décrit dans le chapitre 3 du mémoire, qui entraîne l'incohérence des réponses de ceux-ci. Ce sera à l'utilisateur de la DHT de décider, en fonction de ses besoins et de l'estimation de la propension des noeuds du système à refuser les connexions entrantes, si cette incohérence est négligeable ou non.

Quoiqu'il en soit, lorsque cette incohérence se révèle, le système est, en moyenne, dans un état tel que le pourcentage de réponses aux requêtes transmises tend à être faible, et donc, peut-être, "inacceptable".

Le tableau ci-dessous illustre les résultats obtenus lors des simulations des systèmes Chord selon l'extension proposée. Les interactions entre les noeuds ont été simulées sur les même topologies que celles utilisées dans les simulations de la version initiale de Chord. Le tableau reprend le taux de réponses cohérentes (et incohérentes) aux requêtes, le taux de perte des requêtes, ainsi que le nombre moyen de sauts effectués par les requêtes.

Pourcentage de noeuds inaccessibles	0	10	12	14	16	18	20
Réponses cohérentes (%)	100	100	100	100	100	100	100
Réponses incohérentes (%)	0	0	0	0	0	0	0
Requêtes perdues (%)	0	0	0	0	0	0	0
Nombre de sauts	5,12	5,39	5,43	5,51	5,55	5,61	5,68

Il apparaît clairement, au vu des résultats de ces simulations, que le problème de la perte de messages engendrée par l'inaccessibilité de certains noeuds du système est bien réglé par notre solution. En effet, toutes les requêtes sont résolues, et ce, de façon cohérente.

Le nombre moyen de sauts effectués par les requêtes, lorsque le pourcentage de noeuds inaccessible est nul, est (quasi) identique au nombre moyen de sauts effectués par les requêtes émises, dans les même conditions, dans un système Chord classique. Ceci nous mène à penser que notre solution n'augmente la taille des chemins empruntés par les requêtes que lorsque l'utilisation du mécanisme de proxy est nécessaire.

Le fait que ce mécanisme repose uniquement sur le traitement d'informations contenues dans des messages déjà nécessaires au fonctionnement de Chord, et non sur de l'information de routage transmise dans le but de trouver des routes alternatives vers les noeuds inaccessibles, nous forcent de constater que seule l'augmentation du nombre de sauts effectués par les requêtes constitue un coût supplémentaire au fonctionnement de Chord. La nullité de ce coût, lorsque la topologie du réseau est idéale, et l'évolution linéaire du nombre moyen de sauts effectués par les requêtes nous montrent l'absence d'un quelconque coût fixe à l'utilisation de notre mécanisme de proxy's.

La question est de savoir si cette augmentation de longueur des requêtes est significative. Nous pensons que ce "prix à payer" est négligeable par rapport au gain en terme de fiabilité du système.

Nous allons maintenant analyser les résultats obtenus lors des simulations de la version allégée de notre mécanisme de proxy's. Les environnements d'exécution des simulations sont toujours les mêmes que ceux utilisés dans les simulations précédentes.

Pourcentage de noeuds inaccessibles	0	10	12	14	16	18	20
Réponses cohérentes (%)	100	100	100	100	100	100	100
Réponses incohérentes (%)	0	0	0	0	0	0	0
Requêtes perdues (%)	0	0	0	0	0	0	0
Nombre de sauts	5,11	5,36	5,40	5,47	5,54	5,59	5,67

Encore une fois, notre solution résout, comme annoncé dans la théorie, le problème de pertes de requêtes dues à l'inaccessibilité des noeuds. Dans cet échantillon, la solution allégée semble plus efficace, du point de vue du nombre moyen de sauts effectués par les requêtes, mais la différence n'est pas significative et semble constante.

En comparant le nombre de sauts moyens par requête de nos deux solutions avec le nombre de sauts moyens du système Chord "standard", nous observons que nos solutions ont bien, comme énoncé dans la présentation de celles-ci, des performances identiques à

la solution de base lorsque l'environnement est idéal.

6.3 Travaux futurs

Tout d'abord, remarquons que nous avons simulé le comportement de noeuds fonctionnant selon le mode de proxy présenté au chapitre 4, et ne contenant aucune des améliorations proposées dans celui-ci. Si ces améliorations n'apporteront aucun gain de performance dans des environnements similaires à ceux générés lors de nos simulations, il faudra les prendre en considération lorsque les simulations porteront sur des topologies de réseau plus complexes.

Il nous semble intéressant de poursuivre les simulations en générant des environnements contenant des groupes de noeuds inaccessibles mais dont les constituants peuvent établir des connexions entre eux. Ce type d'environnement pourra (peut-être) nous permettre de distinguer plus significativement les divergences de performance entre les deux solutions proposées.

Faire varier d'autres paramètres que le pourcentage de noeuds inaccessibles (taux de "remplissage" du système, homogénéïté de la répartition des hôtes sur l'espace des identifiants, etc...) pourrait nous permettre d'effectuer un classement permettant de décider quelle solution est la plus adéquate en fonction des caractéristiques attendues de l'environnement d'exécution du système.

Plus simplement, il peut s'avérer utile de "pousser" les caractéristiques des environnements et observer si, au delà d'un certain seuil, les performances d'une des solutions (ou des deux) ne deviennent pas trop "critiques".

Dans [9], les auteurs proposent un autre algorithme de stabilisation. Cet algorithme, plus coûteux, rend Chord plus robuste face aux comportements frauduleux que pourraient avoir certains noeuds du système et qui placeraient celui-ci dans une situation telle que chaque noeud a une connaissance incorrecte de l'identité de son successeur et telle que les procédures de stabilisation ne remédieront pas à cette situation. Il semble que, dans certains cas extrêmes, il soit également possible de placer le système dans un tel état en faisant rentrer un nombre important de noeuds dans le système et en en faisant sortir également un nombre conséquent. Si ce mode est plus robuste, il repose sur l'envoi de requêtes de type *lookup*. Il pourrait donc s'avérer intéressant d'observer la capacité de stabilisation d'un système basé sur cet algorithme, lorsque la réussite de ce genre de requêtes est remise en cause par l'inaccessibilité potentielle des noeuds qui composent ce système.

Il est important de remarquer que les simulations ont été réalisées sur des environnements "statiques". Il reste donc à observer le comportement de nos solutions lorsque des noeuds entrent et sortent du système. Si la faculté d'adaptation du système à l'entrée n'est plus à prouver, cette adaptation n'a pas été étudiée lorsque des noeuds entrent et sortent de manière concurrente. Notons que si la méthode de résolution de nouveau successeur lors d'un départ est adaptée comme énoncé dans la section 4.3, la viabilité du système est assurée. L'étude devra, dans ce cas, porter sur le temps d'adaptation du système à ces départs.

Enfin, nous devons encore observer si notre système ne produit pas des situations telles que certains noeuds deviennent trop vitaux pour la survie du système, et, le cas échéant, nous devons trouver un mécanisme de répartition des tâches de proxy entre les noeuds pouvant rendre ces services.

6.4 Conclusion

Dans ce chapitre, nous avons décrit le type de topologies de réseau simulées lors des simulations. Nous avons également décrit la méthode employée pour constituer les systèmes simulés et défini le nombre de requêtes émises ainsi que leurs caractéristiques.

Ensuite, nous avons présenté les résultats des simulations et conclu que Chord supportait mal l'inaccessibilité des noeuds.

Nous avons montré que notre solution permettait de régler ce problème en introduisant un coût supplémentaire (en terme de nombre de sauts effectués par les requêtes) négligeable par rapport au gain en terme de fiabilité du système.

Nous avons montré que ce coût n'est à supporter que lorsqu'il est nécessaire, étant donné que les performances de notre solution sont identiques à celles de la solution de base lorsque l'environnement est idéal.

Nous n'avons pas pu émettre de jugement de préférence au sujet de nos deux solutions. Il semble que la version allégée offre des performances identiques dans le type d'environnement simulé. Il faudra donc évaluer ces performances dans d'autres types d'environnements afin de voir si, en fait, le choix entre nos deux solutions ne doit pas être fait en fonction des caractéristiques de l'environnement d'exécution du système.

Conclusion

Dans ce mémoire, nous avons présenté les faiblesses potentielles des systèmes Peer-to-Peer actuels. Ensuite, nous avons explicité le principe de DHT en en proposant un modèle général. Nous avons également analysé le principe de fonctionnement de trois DHT spécifiques, décrites dans les termes de notre modèle général.

Après avoir évoqué quelques remarques au sujet de l'applicabilité de ce genre de solutions, nous nous sommes penchés sur une DHT particulière : Chord.

Nous avons proposé, dans un souci de précision, différentes variantes d'implémentation de Chord, toutes respectueuses de sa définition initiale.

L'intérêt de l'identification de ces diverses variantes a été révélé ensuite, lorsque nous avons étudié l'impact de l'inaccessibilité potentielle des noeuds constitutifs d'un système Chord sur sa fiabilité. Nous avons vu que cet impact diffère selon la variante d'implémentation choisie. Nous avons également remarqué que la fiabilité d'un tel système n'est plus assurée lorsque certains noeuds refusent les demandes de connexions émises par leurs pairs. En effet, dans ces conditions d'inaccessibilité, certaines requêtes sont perdues et d'autres résolues de manière incohérente, au vu de la spécification de Chord.

Ces observations ayant été faites, nous avons proposé un système permettant de contrer l'effet de l'inaccessibilité des noeuds sur la fiabilité de Chord. Ce nouveau mécanisme, implanté dans chaque noeud, utilise l'information contenue dans les messages de stabilisation reçus. Il permet, sur base de cette seule information, l'établissement d'une route permettant de transmettre un message vers tout noeud inaccessible, et ce, en profitant des connexions que le noeud inaccessible a lui-même initiées vers d'autres noeuds.

Nous avons également proposé quelques améliorations possibles à notre solution, et nous avons discuté l'impact potentiel sur la performance des noeuds Chord modifiés pour en permettre le support.

Une version "allégée" du protocole a été décrite. Celle-ci garantit la fiabilité et la cohérence du système, tout en limitant la quantité d'informations enregistrée au sein de

chaque noeud.

Ensuite, nous avons présenté les principaux modules d'un simulateur permettant d'observer le comportement de Chord dans un environnement "hostile". Nous avons décrit une représentation de la topologie d'un réseau, ainsi que le comportement des noeuds vérifiant la définition initiale d'un noeud Chord. Nous avons également décrit en détail le comportement des noeuds correspondant à la définition d'un noeud Chord permettant le support des proxy's, véritable clé de voûte de notre solution. Nous avons, ensuite, décrit le comportement d'un noeud vérifiant la définition de la version allégée de notre extension au protocole Chord.

Enfin, les simulations réalisées semblent corroborer les craintes émises lors de l'analyse théorique de l'impact d'une topologie de réseau imparfaite sur le fonctionnement de cette DHT. En effet, le service permettant la résolution du responsable d'une clé n'est manifestement plus assuré lorsque le système n'est pas déployé sur une topologie de réseau idéale.

Cependant, nous avons également montré, lors des simulations, que des techniques simples, comme celle proposée dans ce mémoire, permettent de contrer ce genre de problèmes sans alourdir inutilement le protocole concerné. En effet, nous avons montré par simulation qu'il n'y avait aucune composante fixe dans le coût lié à l'utilisation du mécanisme de proxy. Lorsque le protocole est déployé sur une topologie idéale, les performances de celui-ci sont identiques à celles de la version classique de Chord. Remarquons que ceci peut motiver le choix d'une telle technique lors du déploiement d'une DHT, et ce, même si le risque d'inaccessibilité d'un noeud au sein du système à déployer est faible.

Les simulations réalisées (à l'aide d'un simulateur écrit en Oz, vérifiant la description du simulateur reprise dans ce mémoire) semblent confirmer l'efficacité de la solution lorsque celle-ci est déployée dans un environnement hostile. Le compromis entre efficacité et robustesse semble avoir été trouvé : la solution assure la réussite des requêtes, tout en limitant au plus l'augmentation de la distance parcourue par celles-ci lorsque le réseau n'est pas idéal.

Notons néanmoins que l'apport des différentes améliorations proposées au chapitre 4 du mémoire n'a pas encore été évalué lors des simulations. Ainsi, nous n'avons pas intégré le calcul distribué des plus courts chemins entre les noeuds. Nous n'avons également pas implanté le calcul bi-directionnel des routes les plus courtes.

Il faut également retenir que l'ensemble des topologies possibles n'a pas été totalement couvert lors des simulations. Nous n'avons pas, par exemple, évalué les performances du système, par simulation, lorsque des noeuds ne sont que partiellement inaccessibles.

Aussi, les simulations ont montré que les performances du système supportant les proxy's et les performances de sa variante allégée sont identiques dans les topologies que nous avons considérées. Il semble donc nécessaire de comparer les performances de ces deux solutions dans d'autres types de topologies, afin de déterminer les caractéristiques permettant d'émettre un choix de préférence entre celles-ci.

Pour terminer, la similitude des diverses DHT ayant été établie grâce à l'élaboration d'un modèle général auquel elles correspondent, nous ne pouvons qu'émettre des doutes quant à la correction des autres DHT lorsque celles-ci sont utilisées sur des réseaux hostiles. Le problème d'inaccessibilité des noeuds (et sa solution) doit être transposé au sein du modèle général de DHT afin de déboucher sur une solution applicable aux autres DHT existantes.

Bibliographie

[1] http ://opennap.sourceforge.net/napster.txt .

[2] http ://setiathome.ssl.berkeley.edu/ .

[3] The gnutella protocol specification v0.4
www.people.umass.edu/qifeng/files_web/gnutellaprotocol04.pdf .

[4] www.cisco.com/global/it/media/
presentazioni/infosecurity/infosecurity_perimeter.pdf .

[5] www.cisco.com/warp/public/cc/pd/iosw/ioft/ionetn/prodlit/1195_pp.htm .

[6] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[7] Antony Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218 :329–??, 2001.

[8] P. Maymounkov and D. Mazieres. Kademlia : A peer-to-peer information system based on the xor metric, 2002.

[9] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[10] Jordan Ritter. Why gnutella can't scale. no, really.
www.darkridge.com/ jpr5/doc/gnutella.html .

[11] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network : Properties of large-scale peer-to-peer systems and implications for system design, 2002.

[12] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry : An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[13] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja1, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing.

[14] P. Druschel and A. Rowstron. PAST : A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, pages 75–80, Schloss Elmau, Germany, May 2001.

[15] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[16] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE : The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[17] U.S. Department of Commerce/N.I.S.T. National Technical Information Service. Secure hash standard. In *FIPS 180-1*, April 1995.

[18] www.mozart-oz.org .

[19] P. Krishna Gummadi, Stefan Saroiu, and Steven Gribble. A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems.

Annexe A

Code du simulateur

Le code suivant a été réalisé en Oz [18], l'algorithmique de certaines parties est inspirée d'un simulateur de noeuds Chord réalisé par Bruno Carton[1].

Les différents composants de ce simulateur sont repris ci-dessous. Signalons que les fichiers NodeStandard et NodeproxyLite ne sont repris que partiellement, à raison de leurs différences avec le fichier Nodeproxy.

- *Network.oz* : ce fichier contient le code relatif à l'émulation d'un réseau dont la topologie est configurable.
- *Nodeproxy.oz* : ce fichier contient les algorithmes relatifs au comportement d'un noeud Chord supportant le mécanisme de proxy's.
- *NodeproxyLite.oz* : ce fichier contient les algorithmes relatifs au comportement d'un noeud Chord supportant la variante allégée du mécanisme de proxy's.
- *NodeStandard.oz* : ce fichier contient les algorithmes relatifs au comportement d'un noeud Chord classique.
- *Stats.oz* : ce fichier contient les algorithmes relatifs au comportement d'un noeud spécial dont le but est la récolte des statistiques fournies par les noeuds du système.
- *Test.oz* : ce fichier contient les algorithmes visant à réaliser un nombre quelconque de simulations de systèmes Chord (ou Chord avec proxy's, ou Chord avec proxy's allégé).

[1] bruno.carton@cetic.be

```
1  functor
2  import
3     System(show:Show)
4
5  export
6     insert:Insert %Insertion d'un noeud dans le réseau
7     delete:Delete %Suppression d'un noeud du réseau
8     initMatrix:InitMatrix %Initialisation de la matrice de connectivité.
9     showMatrix:ShowMatrix %Affichage de la matrice de connectivité
10    showMachine:ShowMachine %Affichage des caractéristiques d'une machine (sur base de l'adresse IP de la machine)
11    getMachineSucc:GetMachineSucc %Saisie du successeur d'une machine (sur base de l'adresse IP de la machine)
12
13    setConnectionPossible:SetConnectionPossible %Définition d'un attribut d'accessibilité
14    isConnectionPossible:IsConnectionPossible %Consultation d'un attribut d'accessibilité
15    isConnected:IsConnected %Test sur une connexion entre deux machines du réseau
16    connect:Connect %Connexion de deux machines du réseau
17    disconnect:Disconnect %Déconnexion de deux machines du réseau
18    getState:GetState % Consultation de l'état d'un lien entre deux machines
19    setState:SetState % Mise à jour de l'état d'un lien entre deux machines
20    sendMessage:SendMessage %Envoi d'un message sur le réseau
21    getConnections:GetConnections %Consultation des connexions sortantes actives d'une machine du réseau
22    feedMessage:FeedMessage %Forcer la lecture d'un message par une machine du réseau
23    sendStat:SendStat %Envoi de statistiques au noeud récolteur
24
25 define
26    NetworkSize % Taille maximale du réseau
27    N={Dictionary.new} % Dictionnaire des noeuds
28    ConnectivityMatrix %Matrice d'accesssibilité des noeuds
29    LockNet={Lock.new} % Verrou d'accès au primitives du réseau
30
31    %Enregistrement de l'instance de noeud <Node> à l'adresse ip <Ip> du réseau.
32    proc{Insert Node Ip}
33       if {Dictionary.member N Ip} then
34          {Show Ip#' already used'}
35          raise 'used ip' end
36       else
37          {Dictionary.put N Ip Node}
38       end
39    end
40
41    %Suppression du noeud d'Ip <Ip> du réseau.
42    proc{Delete Ip}
43       PNode
44    in
45       lock LockNet then
46          if {Dictionary.member N Ip}
47          then
48             PNode = {Dictionary.get N Ip}
49             {PNode kill()}
50             {Dictionary.remove N Ip}
```

```
51              if Ip == ~1
52              then
53                 skip
54              else
55
56                 {Loop.'for' 1 NetworkSize 1
57                  proc{$ Index}
58                     {Disconnect Ip Index}
59                     {Disconnect Index Ip}
60                  end
61                 }%loop
62              end%Ip == ~1
63           end%if Ip member N
64        end%lock
65     end%proc
66
67     %Initialisation des conditions d'accès entre les machines. <Size> : Nombre maximal de machine sur le réseau.
68     proc{InitMatrix Size}
69        NetworkSize=Size
70        local
71           Params=state(con_possible:false con_established:false)
72        in
73           {Array.new 1 Size tempidle ConnectivityMatrix}
74            {Loop.'for' 1 NetworkSize 1
75             proc{$ Counter}
76                {Array.put ConnectivityMatrix Counter {Array.new 1 Size Params}}
77            end}
78
79        end%local
80     end
81
82     proc{ShowMatrix}
83        {Loop.'for' 1 NetworkSize 1
84         proc{$ IpSource}
85            if {Dictionary.member N IpSource}
86            then
87               {{Dictionary.get N IpSource} showNode()}
88            end%if
89         end}%loop
90     end%proc
91
92     proc{ShowMachine Ip}
93        if {Dictionary.member N Ip}
94        then
95           {{Dictionary.get N Ip} showNode()}
96        end%Ip présente
97     end%proc
98
99     proc{GetMachineSucc Ip Succ} %Saisie du successeur d'une machine (sur base de l'adresse IP de la machine)
100       if {Dictionary.member N Ip} %Y a-t-il un noeud à cette adresse ?
```

```
      then
         {{Dictionary.get N Ip} getSucc(succ:Succ)}
      end%Ip présente
   end%proc

   %Enregistrement de la possibilité d'établissement (via le booléen Possible) de connexion dans le sens IP1->IP2
   proc{SetConnectionPossible Ip1 Ip2 Possible}
      local State
      in
         if Ip1==Ip2
         then
            {SetState Ip1 Ip2 state(con_possible:false con_established:false)}
         else
            if Possible
            then
               {GetState Ip1 Ip2 State}
               {SetState Ip1 Ip2 state(con_possible:true con_established:State.con_established)}
            else
               {SetState Ip1 Ip2 state(con_possible:false con_established:false)}
            end
         end
      end

   end

   %Res : Une connexion dans le sens IP1->IP2 peut être établie.
   proc{IsConnectionPossible Ip1 Ip2 Res}
      Res={GetState Ip1 Ip2}.con_possible
   end

   %Res : Une connexion est établie dans le sens Ip1->Ip2
   proc{IsConnected Ip1 Ip2 Res}
      Res={GetState Ip1 Ip2}.con_established
   end


   %Etablissement de connexion. Succès si : Deux machines sont enregistrées aux adresses IP1 IP2,
                               %il faut que la connexion soit possible, et qu'une connexion ne soit pas déjà établie dans ce sens.

   proc{Connect Ip1 Ip2}
      local State
      in
         if Ip1==Ip2
         then
            {Show loop_error#Ip1#Ip2}
         end

         if {Bool.'not'{Dictionary.member N Ip1}}
         then
            %raise 'No machine on source IP' end
            skip
```

```
151          end
152
153          {GetState Ip1 Ip2 State}
154          if State.con_possible
155          then
156             if State.con_established
157             then
158                raise 'Already Connected' end
159             else
160                if  {Bool.'not'{Dictionary.member N Ip2}}
161                then
162                 skip
163                 %raise 'No machine on destination IP' end
164                else
165                   {SetState Ip1 Ip2 state(con_possible:true con_established:true) }
166                end
167             end
168
169          else
170             raise 'Host Unreachable' end
171          end
172       end
173    end
174
175
176    %Coupure de la connexion Ip1->Ip2
177    proc{Disconnect Ip1 Ip2}
178       local State
179       in
180          {GetState Ip1 Ip2 State}
181          {SetState Ip1 Ip2 state(con_possible:State.con_possible con_established:false)}
182       end
183    end
184
185
186    %Consultation de l'état du lien entre les deux machines (Ip1->Ip2). State:(con_possible , con_established)
187    %pour connexion possible et connexion établie. (si con_established = true alors con_possible doit valoir true.
188    proc{GetState Ip1 Ip2 State}
189       {Array.get {Array.get ConnectivityMatrix Ip1} Ip2 State}
190    end
191
192    proc{SetState Ip1 Ip2 State}
193       local
194          ArrayAux={Array.get ConnectivityMatrix Ip1}
195       in
196          {Array.put ArrayAux Ip2 State}
197       end
198    end
199
200
```

```
201     %Envoi d'un message de IpSource vers IpDest.
202     %conditions de réussite (càd analyse par IpDest du message émis) : Deux machines présentes à ces IP,
203     %conditions de réussite (càd analyse par IpDest du message émis) : Deux machines présentes à ces IP,
204     %et au moins une connexion établie entre elles.
205     proc{SendMessage IpSource IpDest Message}
206           local Allthere in
207              if {Bool.'not'{Dictionary.member N IpSource}}
208              then
209                 Allthere=false
210              else
211                 if  {Bool.'not'{Dictionary.member N IpDest}}
212                 then
213                    Allthere = false
214                     {Show IpSource#'->'#IpDest#Message}
215                 else
216                    Allthere=true
217                 end%member IpDest
218              end%member IpSource
219
220              if Allthere
221              then
222                 % {Show IpSource#'->'#IpDest#Message.type#Message.source}
223                 if {IsConnected IpSource IpDest}
224                 then
225                    {{Dictionary.get N IpDest} readMessage(fromip:IpSource message:Message)}
226                 else
227                    if {IsConnected IpDest IpSource}
228                    then
229                       {{Dictionary.get N IpDest} readMessage(fromip:IpSource message:Message)}
230                    end
231                 end
232               end%Allthere
233           end%local
234     end%
235
236
237
238
239
240     %Procédures de test.
241
242     proc{GetConnections Ip Connections}
243        {Show getConnectionscalled}
244        {GetConnectionsAux Ip Connections 1}
245     end%proc
246
247     proc{GetConnectionsAux Ip Connections Step}
248        local Aux
249        in
250           if Step=<NetworkSize
```

```
251            then
252               if {IsConnected Ip Step}
253               then
254                  {GetConnectionsAux Ip Aux Step+1}
255                  {List.append Aux [Step] Connections}
256               else
257                  {GetConnectionsAux Ip Connections Step+1}
258               end%if
259            else
260               {List.make 0 Connections}
261            end
262         end%local
263      end%proc
264
265
266      %Procédure nécessaire à la récolte des données statistiques. Permet la lecture d'un message par n'importe quel noeud du réseau
267      %Typiquement : <Message> sera de type lookup.
268      proc{FeedMessage Ip Message}
269         if {Bool.'not'{Dictionary.member N Ip}}
270         then%Aucune machine connectée à cette ip.
271            {Show 'No Machine to feed at IP:'#Ip}
272         else
273            {{Dictionary.get N Ip} readMessage(fromip:Ip message:Message)}
274         end%if
275      end%proc
276
277
278      %Procédure de récolte des statistiques. Envoi vers un noeud spécial voué à la récolte des statistiques
279      %Cette procédure est appelée par un noeud lorsqu'il reçoit une réponse à une requête ou lorsqu'il émet un requête.
280
281      %Pré : un Noeud "récolteur" a été inséré à l'adresse IP <Ip>
282      %     Si la précondition n'est pas respectée, le message est simplement ignoré.
283      %Post : Enregistrement du message par le récolteur.
284
285      proc{SendStat Ip Message}
286         if {Dictionary.member N Ip}
287         then
288            {{Dictionary.get N Ip} readMessage(fromip:Ip message:Message)}
289         else
290            skip
291         end%if
292      end%proc
293
294   end
295
296
297
298
299
300
```

```
1  functor%Foncteur de construction de noeud
2  import
3     System(show:Show showInfo:ShowInfo)
4     OS
5  export
6     newNode:NewNode %Procedure de creation d'instance de noeud.
7  %   node:Node
8  define
9
10    class Node
11       feat
12          Stabilize%Procedure de Stabilisation des noeuds (Exécution périodique)
13          Notify%Procedure d'envoi de message de notification au successeur (Exécution prériodique)
14          BuildFingers%Procedure d'envoi de requêtes de construction de table de routage au successeur (Exécution prériodique)
15          CheckPred%Procedure d'évaluation de la validité du prédécesseur (Exécution périodique)
16          CheckSucc%Procedure d'évaluation de la validité du successeur (Exécution périodique)
17          CheckConnections %Procédure de maintenance des connexions TCP/IP (Exécution périodique)
18          KeepAlive
19
20       attr
21          Id %Identificateur de l'instance du Noeud
22          Ip %Adresse Ip de l'instance du Noeud
23
24          Net %Reseau physique sur lequel le noeud envoie et reçoit ses mesages
25          NetworkSize %Taille du réseau Chord
26
27
28          %%FingerTableSize et PredTableSize doivent correspondre avec le NetworkSize.
29          FingerTableSize %Taille de sa table de routage
30          FingerTable %Table de routage
31
32
33          FingerIndexTable%Table des index des entrées de table de routage (n+1,n+2,n+4,n+8 ...)
34
35          SuccessorListSize
36          SuccessorList%Liste des successeurs du noeud courant(failure recovery)
37
38          PredList%Liste des noeuds ayant envoyé récemment un message notify au noeud courant. (Format : Dictionary)
39
40
41          PredTableSize %Taille de la table des prédecesseurs
42          PredTable % Table des prédecesseurs
43
44
45          PredReceived%Boolean : réception d'une réponse au dernier envoi de requête de prédécesseur.
46
47
48          %Verrous pour l'accès aux diverses tables du noeud.
49          PredTableLock
50          FingerTableLock
```

```
51          SuccessorListLock
52          PredListLock
53
54          %Verrou pour qu'un seul message puisse déclencher un recover successor
55          RecoverLock
56          Connections%Registre des connexions sortantes du noeud
57
58          %Diverses constantes de temps
59          PredUpdate_Time
60          PredUpdate_Rate%Taux maximal de rafraichissement d'un prédécesseur
61          %               (Passé ce taux, le prédécesseur est considéré comme mort).
62
63          RequestPred_Time%Instant du dernier envoi de la demande du prédécesseur du successeur.
64          RequestPredDelay%Delay d'attente de réponse du successeur.
65          %                Au delà, le successeur est considéré comme mort.
66
67
68          ConnectionDelay%Delai de fermeture de connexion non utilisée (en secondes).
69          StabilizeDelay
70          NotifyDelay
71          BuildFingersDelay
72          KeepAliveDelay
73
74
75
76
77
78      meth noop
79         skip
80      end
81
82      meth readMessage(fromip:IP message:Message) %Analyse d'un message reçu
83
84        %Maintenir la connexion vers IP si elle existe
85
86          if {@Net.isConnected @Ip IP}
87            then
88               {self registerConnection(ip:IP)}
89            end
90
91          thread
92             case {Record.label Message.type} of
93                forward
94             then
95                thread
96
97                   if Message.message.type == reply_lookup
98                   then
99                      skip
100                     %{Show @Ip#'Forwarding reply to'#Message.path}
```

```
                end

                try
                   {self connect_and_send(path:Message.path message:Message.message)}
                catch _
                then
                   skip
                end
             end
          []
             lookup
          then
       thread
          local
             Succ
             B4
             Closest_prec_node
             Answer
             ShortestPath
          in
             lock @FingerTableLock then
                lock @SuccessorListLock then
             %{Show 'finger_find_successor detected'}
             %Prendre les références du successeur
                     {Array.get @FingerTable 1 Succ}
                     if Succ==idle
                     then
                        {self shortestPath(path:{List.reverse Message.path} res:ShortestPath)}
                        {Array.put @FingerTable 1 finger(node:node(key:@Id+1 nodeId:Message.source.id nodeIp:Message.source.ip)
                                                         route:ShortestPath)}
                        SuccessorList <-[node(nodeId:Message.source.id nodeIp:Message.source.ip)]
                     else
                        {self eqbefore(low:Message.content.key high:Succ.node.nodeId  res:B4)}
                        {self eqbefore(low:Message.content.key high:Succ.nodeId res:B4)}
               % {Show @Id#eqbefore(low:Message.content.key==@Id}
                 if {Bool.'or' B4 Message.content.key==@Id}
                        then%B4 ==> Le responsable de la clé est trouvé
                   %{Show 'responsible for key'#Message.content.key#'found :'#Succ.node.nodeId}
                        %!!!!Améliorer : Couper les boucles dans Message.path
                           if Message.content.key==@Id
                           then
                              Answer =message(type:reply_lookup
                                              source:node(id:@Id ip:@Ip)
                                              path:{List.reverse Message.path}
                                              content:content(key:Message.content.key key_succ_id:@Id key_succ_ip:@Ip ))

                           else
                              Answer=message(type:reply_lookup
                                             source:node(id:@Id ip:@Ip)
                                             path:{List.reverse {List.append Message.path {List.drop Succ.route 1}}}
                                             content:content(key:Message.content.key
```

```
151                                                    key_succ_id:Succ.node.nodeId
152                                                    key_succ_ip:Succ.node.nodeIp ))
153                        end
154                        if Message.source.id==@Id
155                        then%Le noeud courant est l'initiateur de la requête de recherche.
156                           {self readMessage(fromip:@Ip message:Answer)}
157                        else%Un noeud tiers a initié la requête, envoi de la réponse.
158                           try
159                              {self connect_and_send(path:{List.reverse Message.path} message:Answer)}
160                           catch _
161                           then
162                              skip
163                          %{Show @Id#'Cannot send the Answer to a finger_find_successor request'}
164                          %{self recover_successor(step:1)}
165                          % thread%Attendre que le successeur soit mis à jour et réexaminer le message
166                          %    {Wait 2000}
167                          %    {self readMessage(fromip:IP message:Message)}
168                          % end
169                           end
170                         end%if Message.source.id == @Id
171
172                      else%self eqbefore ==> Le successeur n'est pas responsable de cette clé.
173                         {self closest_preceding_node(id:Message.content.key res:Closest_prec_node)}
174                         if Closest_prec_node==idle
175                         then
176                            {Time.delay 5000}
177                                %{self readMessage(fromip:IP message:Message)}
178                         else
179                            %%ENVOI MESSAGE VERS CLOSEST PRECEDING NODE
180                            local
181                               NewMessage
182                            in
183                               NewMessage=message(type:Message.type
184                                                  source:Message.source
185                                                  path:{List.append Message.path {List.drop Closest_prec_node.route 1}}
186                                                  content:content(key:Message.content.key)
187                                                 )
188                               try
189                                  {self connect_and_send(path:Closest_prec_node.route message:NewMessage)}
190                               catch _
191                               then
192                                 skip
193                               end%catch
194                            end%local
195                         end%Closest_prec_node==idle
196                      end%if self eqbefore
197                   end%Succ==idle
198                end%lock
199             end%lock
200          end%local
```

```
201         end%thread
202              []
203                 reply_lookup
204              then
205                 thread
206                    local
207                       StattoSend
208                    in
209                       StattoSend = message(type:answer
210                                            realsource:node(id:Message.source.id ip:Message.source.ip)
211                                            source:node(id:@Id ip:@Ip)
212                                            %path:Message.path
213                                            pathlength:{List.length Message.path}
214                                            content:content(key:Message.content.key resp:Message.content.key_succ_id))
215                       {@Net.sendStat 0-1 StattoSend}
216                    end%local
217                 end%thread
218
219              []
220                 finger_find_successor
221              then
222                 thread
223                    local
224                       Succ
225                       B4
226                       Closest_prec_node
227                       Answer
228                       ShortestPath
229                    in
230                       lock @FingerTableLock then
231                          lock @SuccessorListLock then
232                       %{Show 'finger_find_successor detected'}
233                       %Prendre les références du successeur
234                              {Array.get @FingerTable 1 Succ}
235                              if Succ==idle
236                              then
237                                 {self shortestPath(path:{List.reverse Message.path} res:ShortestPath)}
238                                 {Array.put @FingerTable 1 finger(node:node(key:@Id+1 nodeId:Message.source.id nodeIp:Message.source.ip)
239                                                                  route:ShortestPath)}
240                                 SuccessorList <-[node(nodeId:Message.source.id nodeIp:Message.source.ip)]
241
242                                 {Time.delay 5000}
243 %                               {self readMessage(fromip:IP message:Message)}
244                              else
245
246                                 {self eqbefore(low:Message.content.key high:Succ.node.nodeId res:B4)}
247                           % {Show @Id#eqbefore(low:Message.content.key high:Succ.nodeId res:B4)}
248                              if B4
249                              then%B4 ==> Le responsable de la clé est trouvé
250                          %{Show 'responsible for key'#Message.content.key#'found :'#Succ.nodeId}
```

```
251                         Answer=message(type:reply_finger_find_successor
252                                        source:node(id:@Id ip:@Ip)
253                                        path:{List.reverse {List.append Message.path {List.drop Succ.route 1}}}
254                                        content:content(key:Message.content.key
255                                                        key_succ_id:Succ.node.nodeId
256                                                        key_succ_ip:Succ.node.nodeIp ))
257                         try
258                            {self connect_and_send(path:{List.reverse Message.path} message:Answer)}
259                         catch _
260                         then
261                            skip
262                         %{Show @Id#'Cannot send the Answer to a finger_find_successor request'}
263                         %{self recover_successor(step:1)}
264                         % thread%Attendre que le successeur soit mis à jour et réexaminer le message
265                         %    {Wait 2000}
266                         %    {self readMessage(fromip:IP message:Message)}
267                         % end
268                          end
269                       else%self eqbefore ==> Le successeur n'est pas responsable de cette clé.
270                          {self closest_preceding_node(id:Message.content.key res:Closest_prec_node)}
271                          if Closest_prec_node==idle
272                          then
273                             {Time.delay 5000}
274 %                           {self readMessage(fromip:IP message:Message)}
275                          else
276                             %%ENVOI MESSAGE VERS CLOSEST PRECEDING NODE
277                             local
278                                NewMessage
279                             in
280                                NewMessage=message(type:Message.type
281                                                   source:Message.source
282                                                   path:{List.append Message.path {List.drop Closest_prec_node.route 1}}
283                                                   content:content(key:Message.content.key)
284                                                  )
285                                try
286                                   {self connect_and_send(path:Closest_prec_node.route message:NewMessage)}
287
288
289                                catch _
290                                then
291                             %    thread %Attendre que la finger table soit à jour et réexaminer le message
292                                    {Time.delay 5000}
293                          %         {self readMessage(fromip:IP message:Message)}
294
295                             %end%thread
296                                end%catch
297                             end%local
298                          end%Closest_prec_node==idle
299                       end%if self eqbefore
300                    end%Succ==idle
```

```
            end%lock
         end%lock
      end%local
   end%thread

[]
   join_find_successor
then
   thread
      local
         Succ %Successeur courant
         Answer

         B4
         Closest_prec_node
      in
      % {Show 'join_find_successor detected'}
         lock @FingerTableLock then
            lock @SuccessorListLock then
               {Array.get @FingerTable 1 Succ}

               if Succ==idle%Aucun successeur connnu pour le noeud courant
               then
               %{Show @Id#'Contact established by a remote node, taking it as successor'}
               %Prise de la source du message comme successeur.
               % Optimisation possible : Prendre le noeud le mieux mis dans la liste
               %                         des noeuds qui ont retransmis la requête
               %
                  {Array.put @FingerTable 1 finger(node:node(key:@Id+1
                                                             nodeId:Message.source.id
                                                             nodeIp:Message.source.ip)
                                                   route:[node(id:@Id ip:@Ip) node(id:Message.source.id
                                                                                   ip:Message.source.ip)])

               }
               SuccessorList <- [node(nodeId:Message.source.id nodeIp:Message.source.ip)]
            % {Show @Id#'Launching maintainance threads'}
               thread
                  {self.Stabilize go}
               end
               thread
                  {self.Notify go}
               end
               thread
                  {self.BuildFingers go}
               end
               thread
                  {self.CheckPred go}
               end
               thread
                  {self.CheckConnections go}
               end
```

```
351                         thread
352                            {self.KeepAlive go}
353                         end
354                         thread
355                            {self.CheckSucc go}
356                         end
357
358
359                      % {Show @Id#'Building the reply' }
360                       %Le noeud se constitue successeur du noeud émetteur.
361                         Answer=message(type:reply_join_find_successor
362                                        source:node(id:@Id ip:@Ip)
363                                        dest:node(id:Message.source.id ip:Message.source.ip)
364                                        path:{List.reverse Message.path}
365                                        content:content(key:Message.content.key key_succ_id:@Id key_succ_ip:@Ip))
366                         try
367                            {self connect_and_send(path:Answer.path message:Answer)}
368                         catch _
369                         then
370                          % {Show @Id#'Cannot send the Answer to a join_find_successor request'}
371                            skip
372                         end
373
374                       else
375
376                         {self eqbefore(low:Message.content.key high:Succ.node.nodeId res:B4)}
377 %                     {Show eqbefore(low:Message.content.key high:Succ.node.nodeId res:B4)}
378                         if B4
379                         then%B4 ==> Le responsable de la clé est trouvé
380                         %{Show @Ip#'Successor found for'#Message.source}
381                            Answer=message(type:reply_join_find_successor
382                                           source:node(id:@Id ip:@Ip)
383                                           dest:node(id:Message.source.id ip:Message.source.ip)
384                                           path:{List.reverse {List.append Message.path {List.drop Succ.route 1}}}
385                                           content:content(key:Message.content.key
386                                                           key_succ_id:Succ.node.nodeId
387                                                           key_succ_ip:Succ.node.nodeIp))
388                            try
389                            %Juste envoyer au dernier émetteur sur le path.
390                            %{Show 'Sending reply to a join request'}
391                               {self connect_and_send(path:{List.reverse Message.path} message:Answer)}
392                            %{Show 'reply to a join request sent'}
393                            catch _
394                            then
395                               skip
396                             % {Show @Id#'Cannot send the Answer to a join_find_successor request'}
397                            end
398                         else%self eqbefore ==> Le successeur n'est pas responsable de cette clé.
399                            {self closest_preceding_node(id:Message.content.key res:Closest_prec_node)}
400                            if Closest_prec_node==idle
```

```
                                then
                                   thread
                                      {Time.delay 2000}
                                   {self readMessage(fromip:IP message:Message)}
                                   end
                                else
                                   %%ENVOI MESSAGE VERS CLOSEST PRECEDING NODE
                                %Ajouter le chemin vers le CLOSEST PRECEDING NODE
                                   local
                                      NewMessage
                                   in
                                      NewMessage=message(type:Message.type
                                                         source:Message.source
                                                         path:{List.append Message.path {List.drop Closest_prec_node.route 1}}
                                                         content:content(key:Message.content.key)
                                                        )
                                      try
                                      %envoyer au CLOSEST PRECEDING NODE
                                         {self connect_and_send(path:Closest_prec_node.route message:NewMessage)}

                                      catch _
                                      then
                                         thread %Attendre que la finger table soit à jour et réexaminer le message
                                            {Time.delay 2000}
                                         {self readMessage(fromip:IP message:Message)}
                                         end%thread
                                      end%catch
                                   end%local
                                end
                             end%if self eqbefore
                          end%if succ==idle
                       end %lock @SuccessorListLock
                    end%lock @FingerTableLock

                 end%local
              end%thread

           []
              reply_join_find_successor
           then
              thread
              %{Show 'reply_join_find_successor detected'}
                 skip
                 lock @FingerTableLock then
                 % {Show 'had one lock'}
                    lock @SuccessorListLock then
                       local
                          ShortestPath
                       in
                    %    {Show 'had second lock'}
```

```
451                         {self shortestPath(path:{List.reverse Message.path} res:ShortestPath)}
452                         {Array.put @FingerTable 1 finger(node:node(key:@Id+1
453                                                                    nodeId:Message.content.key_succ_id
454                                                                    nodeIp:Message.content.key_succ_ip)
455                                                                    route:ShortestPath)}
456                         SuccessorList <-[node(nodeId:Message.content.key_succ_id
457                                               nodeIp:Message.content.key_succ_ip)]
458                      end
459                   %{Show 'Launching threads'}
460                      thread
461                         {self.Stabilize go}
462                      end
463                      thread
464                         {self.Notify go}
465                      end
466                      thread
467                         {self.BuildFingers go}
468                      end
469                      thread
470                         {self.CheckPred go}
471                      end
472                      thread
473                         {self.CheckConnections go}
474                      end
475                      thread
476                         {self.KeepAlive go}
477                      end
478                      thread
479                         {self.CheckSucc go}
480                      end
481   %                 {Show 'Threads lauched'}
482                     end%lock
483                  end%lock
484               end
485
486            []
487               reply_finger_find_successor
488            then
489               thread
490                  %Trouver l'index correspondant au message
491                  local
492                     Index_to_update
493                  %!!!!Empêcher cette proc de boucler si cle invalide...
494                     FindEntry=proc {$ Index Cle Res}
495                                   if {Array.get @FingerIndexTable Index}==Cle
496                                   then
497                                      Res=Index
498                                   else
499                                      {FindEntry Index+1 Cle Index_to_update}
500                                   end%Index==Cle
```

```
                        end
                in
                   try
                      Index_to_update={FindEntry 1 Message.content.key}
                   %Remplacer l'entrée correspondante
                      lock @FingerTableLock then
                         if Message.content.key_succ_id==@Id %
                         then%Le noeud se situe virtuellement dans sa propre table de routage,
                              %insérer idle pour éviter le bouclage des transferts.
                            {Array.put @FingerTable Index_to_update idle}
                         else
                            local
                               ShortestPath
                            in
                               {self shortestPath(path:{List.reverse Message.path} res:ShortestPath)}
                               {Array.put @FingerTable Index_to_update finger(node:node(key:Message.content.key
                                                                                        nodeId:Message.content.key_succ_id
                                                                                        nodeIp:Message.content.key_succ_ip)
                                                                              route:ShortestPath)}


                            end%local
                         end% Message.content.key_succ_id==@Id %
                      end%lock
                   catch _
                   then
                      skip
                   end%try


                end%local
             end
          []
             request_predecessor
          then
             thread
                local
                   Pred
                   Mess
                in
                   {Array.get @PredTable 1 Pred}
                   if Pred==idle
                   then
                      {Time.delay 2000}
                      {self readMessage(fromip:IP message:Message)}
                   else
                      Mess=message(type:reply_predecessor
                                   source:node(id:@Id ip:@Ip)
                                   path:{List.reverse {List.append Message.path {List.drop Pred.route 1}}}
                                   content:content(pred_id:Pred.node.nodeId pred_ip:Pred.node.nodeIp succ_list:@SuccessorList))

                      try
                      {Show @Ip#'reply_predecessor sent'}
%
```

```
551 %                     {Show {List.reverse Message.path} }
552                          {self connect_and_send(path:{List.reverse Message.path}
553                                                 message:Mess)
554                          }
555                       catch _
556                       then
557                          skip
558                       end
559                    end%if Pred==idle
560                 end%local
561              end
562           []
563              reply_predecessor
564              %Le message contient le prédecesseur du successeur et sa succList
565           then
566              thread
567
568                 local
569                    BetterSucc
570                    Succ = {Array.get @FingerTable 1}
571                  %  UpdatedSucc
572 %                  Mess
573                    ShortestPath
574                 in
575 %                  {Show @Ip#'set PredReceived to true'}
576                    PredReceived <-true
577                    lock @FingerTableLock then
578                 %traitement du prédecesseur du successeur
579                       if Succ==idle
580                       then
581                          skip %Failure Recovery
582                       else
583                          if {Bool.'not' Message.content.pred_id==@Id}%le pred du succ n'est pas moi
584                          then
585                             {self eqbefore(low:Message.content.pred_id  high:Succ.node.nodeId res:BetterSucc)}
586                             if BetterSucc
587                             then
588                                {self shortestPath(path:{List.reverse Message.path} res:ShortestPath)}
589                                {Array.put @FingerTable 1 finger(node:node(key:Succ.node.key
590                                                                           nodeId:Message.content.pred_id
591                                                                           nodeIp:Message.content.pred_ip)
592                                                                 route:ShortestPath)
593                                }
594
595                             end%if BetterSucc
596                          else%Le predecesseur du successeur du noeud courant est le noeud courant lui-même
597                          %traitement de la succList (Utiliser sa succlist dans ce cas uniquement.
598                          %sinon, attendre la succlist du nouveau successeur
599                          %==>Etablir une liste de longueur max SuccessorListSize avec le successeur
600                          %   comme tête de liste et sa SuccessorList comme queue.
```

```
%Conservation de l'ancien successeur, mise à jour du chemin vers celui-ci.(Il existe peut-être un chemin plus court)
                              local
                                 ShortestPathtoSucc
                              in
                                 {self shortestPath(path:Succ.route res:ShortestPathtoSucc)}
                                 {Array.put @FingerTable 1 finger(node:node(key:Succ.node.key
                                                                            nodeId:Succ.node.nodeId
                                                                            nodeIp:Succ.node.nodeIp)
                                                                  route:ShortestPathtoSucc)

                                 }
                              end


                              lock @SuccessorListLock then
SuccessorList <- {List.append [node(nodeId:Succ.node.nodeId nodeIp:Succ.node.nodeIp)] {List.take Message.content.succ_list @SuccessorL
                                 SuccessorList <- {List.filter @SuccessorList fun {$ Item}
                                                                 {Bool.'not' Item.nodeId==@Id}
                                                               end

                                                  }
                              end%lock
                           end%if bool.'not'...

                        end%succ==idle
                     end%lock


                  end%local
               end%thread
            []
               notify
            then
               thread
                  local
                     CurrentTime = {OS.time}
                     Pred = {Array.get @PredTable 1}
                     BetterPred

                  in
                     lock @PredListLock then
                        local ShortestPath
                        in
                           {self shortestPath(path:{List.reverse Message.path}
                                              res:ShortestPath)}
                           {Dictionary.put @PredList Message.source.id predinfo(time:CurrentTime
                                                                                node:Message.source
                                                                                route:ShortestPath)}

                        end%local
                     end%lock
```

```
651                  lock @PredTableLock then
652                     if Pred\=idle
653                     then
654                        {self eqbefore(low:Message.source.id high:@Id-1
655                                       place:Pred.node.nodeId res:BetterPred)}
656                        if BetterPred
657                        then
658                           local ShortestPath
659                           in
660                              {self shortestPath(path:{List.reverse Message.path}
661                                                 res:ShortestPath)}
662                              {Array.put @PredTable 1 finger(node:node(nodeId:Message.source.id
663                                                                       nodeIp:Message.source.ip)
664                                                             route:ShortestPath
665                                                            )}
666                              lock @PredListLock then
667                                 skip
668                                 local
669                                    Answer
670                                 in
671                                    Answer=message(type:reply_notify
672                                                   source:node(id:@Id ip:@Ip)
673                                                   path:ShortestPath
674                                                   content:content(predlist:{Dictionary.items @PredList})
675                                                  )
676                                    {self connect_and_send(path:ShortestPath
677                                                           message:Answer)}
678
679                                 end%local
680                              end%lock
681                           end%local
682                           PredUpdate_Time<-{OS.time}
683                        end%if BetterPred
684
685                        if (Pred.node.nodeId==Message.source.id)
686                        then
687                           PredUpdate_Time<-{OS.time}
688                        end
689
690
691
692                     else%Pred==idle
693                        local ShortestPath
694                        in
695                           {self shortestPath(path:{List.reverse Message.path}
696                                              res:ShortestPath)}
697                           {Array.put @PredTable 1  finger(node:node(nodeId:Message.source.id
698                                                                     nodeIp:Message.source.ip)
699                                                           route:ShortestPath
700                                                          )
```

```
                                }
                             end%local
                             PredUpdate_Time<-{OS.time}
                          end%Pred\=idle
                       end%lock
                    end%local
                 end
              []
                 reply_notify
              then
                 thread
                   %Parcourir la liste des prédécesseurs reçus,
                   {Loop.'for' 1 {List.length Message.content.predlist} 1
                    proc{$ Index}
                       if  {List.nth Message.content.predlist Index}==idle
                       then
                          {Show probidledansmessage}
                       end

                       if (@Id \= {List.nth Message.content.predlist Index}.node.id)
                       then
                          %{Show @Id}
                          %.{Show {List.nth Message.content.predlist Index}}
                          local
                             ShortestPath
                             MaximalPathtothisPred = {List.append {List.reverse Message.path}
                                                                  {List.drop {List.nth Message.content.predlist Index}.route  1}}

                             Answer
                             ThisIsBetterSucc
                          in
%Le noeud courant est-il entre l'entree n°Index de la predlist reçue et l'émetteur du reply_notify ?
                            {self eqbefore(low:@Id
                                           high:Message.source.id
                                           place:{List.nth Message.content.predlist Index}.node.id
                                           res:ThisIsBetterSucc)}
                       %Si oui, alors se faire connaitre de cette Entree, et se proposer comme meilleur Successeur
                                if ThisIsBetterSucc
                                then
                                   {self shortestPath(path:MaximalPathtothisPred
                                                      res:ShortestPath )}

                                   Answer=message(type:better_succ
                                                  source:node(id:@Id ip:@Ip)
                                                  path:ShortestPath
                                                  content:content()
                                                 )
                                   {self connect_and_send(path:ShortestPath
                                                          message:Answer)}

                                end% ThisIsBetterSucc
                             end%local
```

```
751                  end%Id\= pred.id
752               end%proc
753            }
754         end%thread
755
756
757      []
758         reply_find_successor
759      then
760         thread
761            {Show 'Find successor found reply :'#Message}
762         end
763
764      []
765         better_succ
766      then
767         skip
768         %regarder si le successeur est vraiment meilleur, si oui, remplacer le successeur.
769         local
770            BetterSucc
771            Succ
772            ShortestPath
773         in
774            lock @FingerTableLock
775            then
776               {Array.get @FingerTable 1 Succ}
777               if Succ \= idle
778               then
779                  {self eqbefore(low:Message.source.id  high:Succ.node.nodeId-1 res:BetterSucc)}
780                  if BetterSucc
781                  then
782                  %{Show @Id#'BetterSuccessor found'#Message.source.id}
783                     {self shortestPath(path:{List.reverse Message.path}
784                                        res:ShortestPath)}
785                     {Array.put @FingerTable 1 finger(node:node(key:@Id+1
786                                                                nodeId:Message.source.id
787                                                                nodeIp:Message.source.ip)
788                                                      route:ShortestPath)}
789                  else
790                  %{Show @Id#'Successor suggested is not better'#Message.source.id}
791                     skip
792                  end%if BetterSucc
793               else%Succ==idle
794                  {self shortestPath(path:{List.reverse Message.path}
795                                     res:ShortestPath)}
796                  {Array.put @FingerTable 1 finger(node:node(key:@Id+1
797                                                             nodeId:Message.source.id
798                                                             nodeIp:Message.source.ip)
799                                                   route:ShortestPath)}
800               end
```

```
                end%Lock
             end%local
          []
             keep_alive
          then
             skip

          else%case
             {Show 'Unrecognized message'#@Id#@Ip}
             {Show Message}
             skip
          end%case

       end%thread
    end

    %Initialisation du noeud

    meth init(ip:IP id:ID network:Network networkSize:NetworkS
              fingerTableSize:FingerTableS
              predTableSize:PredTableS
              successorListSize:SuccessorListS  %Taille des tables du noeud
              stabilizeDelay:StabilizeD <=250
              notifyDelay:NotifyD <=250
              buildFingersDelay:BuildFingersD<=10000
              keepAliveDelay:KeepAliveD<=10000 %Fréquence d'application des primitives de stab.
              connectionDelay:ConnectionD <=20 %Temps maximum d'inactivité d'une connexion (avant fermeture, en secondes)
              requestPredDelay:RequestPredD <= 250
             )
       RecoverLock <-{Lock.new}
       PredTableLock <-{Lock.new}
       FingerTableLock <- {Lock.new}
       SuccessorListLock <-{Lock.new}
       PredListLock <-{Lock.new}

       {Show 'Enregistrement des parametres du noeud et du systeme chord'}
%      %Enregistrement des parametres du noeud et du systeme chord
       NetworkSize<-NetworkS

       Id<-ID
       Ip<-IP
       FingerTableSize<-FingerTableS
       PredTableSize<-PredTableS
       SuccessorListSize<-SuccessorListS
       PredUpdate_Rate<-6
       PredUpdate_Time<-{OS.time}

       ConnectionDelay<-ConnectionD
       StabilizeDelay <-StabilizeD
```

```
851         NotifyDelay<-NotifyD
852         BuildFingersDelay<-BuildFingersD
853         KeepAliveDelay<-KeepAliveD
854         RequestPredDelay<-RequestPredD
855
856
857         PredReceived<- true
858 %       {Show 'Enregistrement dans la couche réseau'}
859         %Enregistrement dans la couche réseau
860         Net<-Network
861 %       {Show 'ip inserted:'#@Ip}
862         {@Net.insert self @Ip}
863 %        {Show 'Initialisation des tables de routage'}
864         %Initialisation des tables de routage
865         {Array.new 1 @FingerTableSize idle @FingerTable}
866         {Array.new 1 @PredTableSize idle @PredTable}
867         {List.make 0 @SuccessorList}
868         {Dictionary.new @PredList}
869         {Dictionary.new @Connections}
870
871
872         {Array.new 1 @FingerTableSize idle @FingerIndexTable}
873         {Loop.'for' 1 @FingerTableSize 1
874          proc{$ Index}
875             local
876                Avancement = {Number.'+' @Id {Number.pow 2 {Number.'-' Index 1} }}
877                KeyToSearch
878             in
879                KeyToSearch = {Int.'mod' Avancement @NetworkSize}
880                {Array.put @FingerIndexTable Index KeyToSearch}
881             end
882          end
883         }
884
885
886         %Procédure de stabilisation
887         self.Stabilize = {New Time.repeat
888                           setRepAll(action: proc {$}
889                                                local
890                                                   Succ
891                                                   ShortestPath
892                                                in
893                                                   {Array.get @FingerTable 1 Succ}
894                                                   if Succ==idle
895                                                   then
896                                                      %%RECOVERY
897                                                      %{Show 'stabilize recovery'}
898                                                      {self recover_successor()}
899                                                   else
900                                                      if @PredReceived
```

```
                                                then
                                                   RequestPred_Time <- {OS.time}
                                                   {self shortestPath(path:Succ.route res:ShortestPath)}
                                                   lock @FingerTableLock
                                                   then
                                                      {Array.put @FingerTable 1 finger(node:node(key:Succ.node.key
                                                                                                 nodeId:Succ.node.nodeId
                                                                                                 nodeIp:Succ.node.nodeIp)
                                                                                       route:ShortestPath)}

                                                   end%lock
                                                   {self connect_and_send(path:ShortestPath
                                                                          message:message(type:request_predecessor
                                                                                          path:ShortestPath
                                                                                          source:node(id:@Id ip:@Ip)
                                                                                          content:content()))}

                                                    PredReceived<-false
                                                end
                                             end%if Succ==idle
                                          end%local
                                       end%proc
                                delay: @StabilizeDelay
                                number: ~1)
                     }


        self.Notify = {New Time.repeat
                       setRepAll(action: proc {$}
                                            local
                                               Succ
                                            in
                                               lock @FingerTableLock then
                                                  {Array.get @FingerTable 1 Succ}
                                                  if Succ==idle
                                                  then
                                                     %%RECOVERY
                                                     {self recover_successor()}
                                                  else

                                                     try
                                                        {Show 'Notify sent by'#@Id#'to'#Succ}
%                                                       {self connect_and_send(path:Succ.route
                                                                               message:message(type:notify
                                                                                               path:Succ.route
                                                                                               source:node(id:@Id ip:@Ip)
                                                                                               content:content()))}


                                                     catch _ %ErreurEnvoi
```

```
                                                then%
                                                   %{Show 'rputpur'#ErreurEnvoi}
                                                   %{self recover_successor()} non !
                                                skip
                                             end
                                          end%if Succ==idle
                                       end%lock
                                    end%local
                                 end%proc
                              delay: @NotifyDelay
                              number: ~1)
                     }


         self.BuildFingers = {New Time.repeat
                              setRepAll(action: proc {$}
                                           local
                                              Succ
                                           in
                                              {Array.get @FingerTable 1 Succ}
                                              if Succ==idle
                                              then
                                                 %%RECOVERY
                                                 {self recover_successor()}
                                              else
                                                 {self build_fingers(contactIp:Succ.node.nodeIp
                                                                     contactId:Succ.node.nodeId
                                                                     contactPath:Succ.route
                                                                    )}
                                              end%if Succ==idle
                                           end%local
                                        end%proc
                                     delay: @BuildFingersDelay
                                     number: ~1)
                           }


         self.CheckPred = {

                           New Time.repeat
                           setRepAll(action: proc {$}
                                        local
                                           Pred
                                           Diff_Time
                                        in
                                           {Array.get @PredTable 1 Pred}
                                           if Pred==idle
                                           then
```

```
                                                skip%Wait for a notify
                                             else
                                                Diff_Time={OS.time}-@PredUpdate_Time
                                                if {Value.'>=' Diff_Time @PredUpdate_Rate}
                                                then
                                                   {Array.put @PredTable 1 idle}
                                                end%=>Pred
                                             end%if Succ==idle

                                             lock @PredListLock
                                             then
                                                local
                                                   PredL={Dictionary.items @PredList}
                                                in
                                                   {Loop.'for' 1 {List.length PredL} 1
                                                    proc{$ Index}
                                                       if ({OS.time} > ({List.nth PredL Index}.time
                                                                        + 2*{Int.'div' @NotifyDelay 1000}))

                                                       then
                                                            %Pas de notify envoyé depuis 2*le delai maximal
                                                            %d'envoi de notify, enlever ce noeud des prédécesseurs
                                                          {Dictionary.remove @PredList {List.nth PredL Index}.node.id}
                                                       end
                                                    end%proc
                                                   }
                                                end%local
                                             end%lock
                                          end%local
                                       end%proc
                                    delay: 1000
                                    number: ~1)
                        }

   self.CheckConnections = {
                           New Time.repeat
                           setRepAll(action: proc {$}
                                                local
                                                   CurrentConnections = {Dictionary.entries @Connections}
                                                   LocalTime = {OS.time}
                                                in
                                                   {Loop.'for' 1 {List.length CurrentConnections} 1
                                                    proc{$ Index}
                                                       if LocalTime >= {List.nth CurrentConnections Index}.2+@ConnectionDelay
                                                       then
                                                          %Aucun message transmis entre ces deux noeuds depuis plus de
                                                          % @ConnectionDelay secondes, terminer la connexion
                                                          try
                                                             {@Net.disconnect @Ip {List.nth CurrentConnections Index}.1}
                                                             {Dictionary.remove @Connections
                                                                                {List.nth CurrentConnections Index}.1}
```

```
1051                                                                    catch _
1052                                                                    then
1053                                                                       skip
1054                                                                    end
1055
1056                                                                 end
1057
1058
1059                                                              end%proc
1060                                                             }
1061                                                          end%local
1062                                                       end%proc
1063                                                delay: 1000
1064                                                number: ~1)
1065                                          }
1066          self.KeepAlive = {
1067                           New Time.repeat
1068                           setRepAll(action: proc {$}
1069                                                lock @FingerTableLock
1070                                                then
1071                                                   {Loop.'for' 1 @FingerTableSize 1
1072                                                    proc{$ Index}
1073                                                       local Entree KEEP
1074                                                       in
1075                                                          {Array.get @FingerTable Index Entree}
1076                                                          if {Bool.'not' Entree==idle}
1077                                                          then
1078                                                             KEEP=message(type:keep_alive
1079                                                                          source:node(id:@Id ip:@Ip)
1080                                                                          path:Entree.route
1081                                                                          content:content())
1082  %                                                                    {Show @Ip#'sending keepalive to'#Entree.route}
1083                                                                       %Tentative de connexion  avec le nexthop de
1084                                                                       %l'entree dans la table de routage
1085                                                             try
1086                                                                {@Net.connect @Ip {List.nth Entree.route 2}.ip}
1087                                                                {self registerConnection(ip:{List.nth Entree.route 2}.ip)}
1088                                                             catch ConnectionError
1089                                                             then
1090                                                                if ConnectionError=='Already Connected'
1091                                                                then
1092                                                                   {self registerConnection(ip:{List.nth Entree.route 2}.ip)}
1093                                                                end%if ConnectionError ...
1094                                                             end%catch
1095                                                             try
1096                                                                {self connect_and_send(path:Entree.route message:KEEP)}
1097                                                             catch _
1098                                                             then
1099                                                                      %La route n'est plus valide, supprimer cette entrée.
1100                                                                {Array.put @FingerTable Index idle}
```

```
                end
             else
                skip
             end%Entree\=idle
          end%local
       end%proc
    }%loop
            %!!!!! Meme comportement avec le prédécesseur.

    lock @PredTableLock
    then
       local
          Pred={Array.get @PredTable 1}
          KEEP
       in
          if {Bool.'not' Pred==idle}
          then
             KEEP=message(type:keep_alive
                          source:node(id:@Id ip:@Ip)
                          path:Pred.route
                          content:content())
                         %Tentative de connexion avec le nexthop vers le prédécesseur
             try
                {@Net.connect @Ip {List.nth Pred.route 2}.ip}
                {self registerConnection(ip:{List.nth Pred.route 2}.ip)}
             catch ConnectionError
             then
                if ConnectionError=='Already Connected'
                then
                   {self registerConnection(ip:{List.nth Pred.route 2}.ip)}
                else
                   skip
                                %{Show @Ip#ConnectionError}
                end%if ConnectionError ...


             end%catch
             try
                {self connect_and_send(path:Pred.route message:KEEP)}
             catch _
             then
                                %La route n'est plus valide, supprimer cette entrée.
                {Array.put @PredTable 1 idle}
                                %!!!!recover_predecessor
             end

          end%if Pred \=idle

       end%local
    end%lock
```

```
                                        end%lock


                                    end%proc
                                delay: @KeepAliveDelay
                                number: ~1)
                        }


      self.CheckSucc  = {

                         New Time.repeat
                         setRepAll(action: proc {$}
                                              local
                                                 Diff_Time
                                              in
                                                 if {Bool.'not' @PredReceived}
                                                 then
                                                    Diff_Time={OS.time}-@RequestPred_Time
                                                    if {Value.'>=' Diff_Time @RequestPredDelay}
                                                    then
                                                       %{Show @Id#@Ip#'Successor Timed out, setting it to idle...'}
                                                       {Array.put @FingerTable 1 idle}
                                                       PredReceived <- true
                                                       {self recover_successor()}
                                                    end%=>
                                                 end
                                              end%local
                                           end%proc
                                delay: 500
                                number: ~1)
                         }

        % {Show 'Fin'}
    end

    %Affichage des statistiques du noeud
    meth showNode()
       %thread
       {Show node(ip:@Ip id:@Id)}
       {Loop.'for' 1 @FingerTableSize 1
        proc{$ Index}
           local Avancement = {Number.pow 2 {Number.'-' Index 1} }
           in

              if {Array.get @FingerTable Index}\= idle
              then
                 {Show 'FingerTable@'#Index#':N'#@Id#'+'#Avancement#':' #{Array.get @FingerTable Index}.node}
```

```
1201                    {Show 'Route to this finger:'#{Array.get @FingerTable Index}.route}
1202                 else
1203                    {Show 'FingerTable@'#Index#':N'#@Id#'+'#Avancement#':' #{Array.get @FingerTable Index}}
1204                 end
1205              end%local
1206           end%proc
1207        }
1208
1209        if {Array.get @PredTable 1}== idle
1210        then
1211           {Show 'Pred : '#{Array.get @PredTable 1}}
1212        else
1213           {Show 'Pred : '#{Array.get @PredTable 1}.node}
1214           {Show 'Route : '#{Array.get @PredTable 1}.route}
1215        end
1216        {Show 'SuccessorList contains '#{List.length @SuccessorList}#'elements'}
1217        /*
1218        {Loop.'for' 1 {List.length @SuccessorList} 1
1219         proc{$ Index}
1220            {Show 'Successor number'#Index#':'#{List.nth @SuccessorList Index}}
1221         end%proc
1222        }
1223        */
1224        {Show 'PredList :'}
1225        local PredL={Dictionary.items @PredList}
1226        in
1227           {Loop.'for' 1 {List.length PredL} 1
1228            proc{$ Index}
1229               {Show {List.nth PredL Index}}
1230            end%proc
1231           }
1232        end
1233 %         {Show 'PredList contains '#{List.length @PredList}#'elements'}
1234        /*
1235        {Loop.'for' 1 {List.length @PredList} 1
1236         proc{$ Index}
1237            {Show 'Predecessor number'#Index#':'#{List.nth @PredList Index}}
1238         end%proc
1239        }
1240        */
1241        {Show 'Active tcp/ip connections :'}
1242        {Show {Dictionary.entries @Connections}}
1243
1244        %end%thread
1245     end
1246
1247     meth getSucc(succ:Succ)%Retourne l'ID du successeur (-1 si pas de successeur courant)
1248        lock @FingerTableLock then
1249           local
1250              Successeur =  {Array.get @FingerTable 1}
```

```
1251            in
1252               if Successeur == idle
1253               then
1254                  Succ=0-1
1255               else
1256                  Succ=Successeur.node.nodeId
1257               end %Successeur ==idle
1258            end%local
1259         end%lock
1260      end
1261
1262
1263
1264      %Connexion au contact et demande du successeur(1ère partie du Join)
1265      meth join(contactIp:ContactIP contactId:ContactID)
1266         local Connecte Message
1267         in
1268            try
1269            %Etablissement de la connexion avec le contact
1270               {@Net.connect @Ip ContactIP}
1271               Connecte=true
1272            catch ErreurConnection
1273            then
1274               if ErreurConnection=='Already Connected'
1275               then
1276                  Connecte=true
1277               else
1278                  Connecte=false
1279               end
1280            end
1281
1282            if Connecte
1283            then%La connexion avec le contact est établie, envoi de la requête
1284               try
1285                  %Enregistrer la connection dans le connection registry
1286                  {self registerConnection(ip:ContactIP)}
1287                  Message=message(type:join_find_successor
1288                                  source:node(id:@Id ip:@Ip)
1289                                  path:[node(id:@Id ip:@Ip)
1290                                        node(id:ContactID ip:ContactIP)]
1291                                  content:content(key:@Id+1))
1292                  {@Net.sendMessage @Ip ContactIP Message}
1293               catch ErreurEnvoi
1294               then
1295                  {Show 'Error while sending message at join:'#ErreurEnvoi}
1296                  % {Show ErreurEnvoi}
1297               end
1298
1299            else%La connexion a echoué.
1300               {Show 'Connection failed with contact'}
```

```
1301            end
1302         end
1303      end
1304
1305
1306      meth build_fingers(contactIp:ContactIP contactId:ContactID contactPath:ContactPath)
1307         local Succ
1308         in
1309            lock @FingerTableLock then
1310               Succ={Array.get @FingerTable 1}
1311               if {Bool.'not' Succ==idle}
1312               then
1313                  try
1314                     {Loop.'for' 2 @FingerTableSize 1
1315                      proc{$ Index}
1316                         local
1317                            Avancement = {Number.'+' @Id {Number.pow 2 {Number.'-' Index 1} }}
1318                            KeyToSearch = {Int.'mod' Avancement @NetworkSize}
1319                            SuccResponsability
1320                         in
1321                            {self eqbefore(low:KeyToSearch
1322                                           high:{Array.get @FingerTable 1}.node.nodeId
1323                                           res:SuccResponsability)}
1324                            if SuccResponsability
1325                            then
1326                               {Array.put @FingerTable Index finger(node:node(key:KeyToSearch
1327                                                                              nodeId:Succ.node.nodeId
1328                                                                              nodeIp:Succ.node.nodeIp)
1329                                                                    route:{List.take Succ.route {List.length Succ.route}}
1330                                                                   )
1331
1332                               }
1333                            else
1334                               % {Show 'KeytoSearch'#Avancement#'mod'#@NetworkSize#'='#KeyToSearch}
1335                               try
1336                                  {self connect_and_send(path:Succ.route  message:message(type:finger_find_successor
1337                                                                                          path:Succ.route
1338                                                                                          source:node(id:@Id ip:@Ip)
1339                                                                                          content:content(key:KeyToSearch ) ))}
1340
1341
1342                               catch _ %ErreurEnvoi
1343                               then
1344                                  {self recover_successor()}
1345                               end%catch ErreurEnvoi
1346
1347                            end
1348                         end
1349                      end
1350                     }
                     catch ErreurEnvoi
```

```
1351                  then
1352                     {Show 'Error while sending message at fingers building:'#ErreurEnvoi}
1353                     {Show ErreurEnvoi}
1354                  end
1355               end%if Succ==idle
1356            end%lock
1357         end%local
1358      end %method
1359
1360      meth eqbefore(low:L high:H place:P <= @Id res:R)
1361         LAux = {Int.'mod' L + @NetworkSize - P @NetworkSize}
1362         HAux = {Int.'mod' H + @NetworkSize - P @NetworkSize}
1363      in
1364         R = LAux =< HAux
1365      end
1366
1367      meth closest_preceding_node(id:ID res:Res)
1368         local
1369            PrecedingList={Cell.new {List.make 0}}
1370            {Cell.assign PrecedingList {List.append {Cell.access PrecedingList} [idle]}}
1371            ResultList
1372         in
1373            lock @FingerTableLock then
1374               {Loop.'for' 1 @FingerTableSize 1
1375                proc{$ Index}
1376                   local Entree B4 NotMe
1377                   in
1378                      {Array.get @FingerTable Index Entree}
1379                      if {Bool.'not' Entree==idle}
1380                      then
1381                         {self eqbefore(low:Entree.node.nodeId high:ID res:B4)}
1382                         NotMe = Entree.node.nodeId\=@Id
1383                         if {Bool.and B4 NotMe}
1384                         then
1385                            {Cell.assign PrecedingList {List.append {Cell.access PrecedingList} [Entree]}}
1386                         else
1387                            skip
1388                         end%if
1389                      end%Entree\=idle
1390                   end%local
1391                end%proc
1392               }%loop
1393               ResultList={Cell.access PrecedingList}
1394               {List.nth ResultList {List.length ResultList} Res}
1395            end%lock
1396         end %local
1397      end %method closest_preceding_node(id:Id)
1398
1399      meth connect_and_send(path:Path message:Message)
1400         local Message2Send
```

```
1401        in
1402           if {List.length Path}==2
1403           then
1404              %Le nexthop est le destinataire final du message.
1405              Message2Send=Message
1406           else
1407              %le nexthop fait figure de proxy
1408              Message2Send=message(type:forward source:node(ip:@Ip id:@Id) path:{List.drop Path 1} message:Message)
1409           end %if length==2
1410           local
1411              Ipdest={List.nth Path 2}.ip
1412              Connected_here_to_there = {@Net.isConnected @Ip Ipdest}
1413              Connected_there_to_here = {@Net.isConnected Ipdest @Ip}
1414              Connecte={Bool.'or' Connected_here_to_there Connected_there_to_here }
1415           in
1416              if Connecte
1417              then
1418                 if Connected_here_to_there
1419                 then
1420                    %Réinitialiser le timer sur la connexion vers Ipdest
1421                    {self registerConnection(ip:Ipdest)}
1422                 end%Connected_here_to_there
1423
1424                 try
1425                    {@Net.sendMessage @Ip Ipdest Message2Send}
1426                 catch EnvoiExc
1427                 then
1428
1429                    raise EnvoiExc end
1430                 end
1431              else
1432                 try
1433                    {@Net.connect @Ip Ipdest}
1434                    %Enregistrer cette connection dans Connection registry
1435                    {self registerConnection(ip:Ipdest)}
1436                    {@Net.sendMessage @Ip Ipdest Message2Send}
1437                 catch EnvoiExc
1438                 then
1439                    if EnvoiExc=='Already Connected'
1440                    then
1441                       {self registerConnection(ip:Ipdest)}
1442                       try
1443                          {@Net.sendMessage @Ip Ipdest Message2Send}
1444                       catch Ex
1445                       then
1446                          raise Ex end
1447                       end
1448                    else
1449                       %{Show 'Connection error, message won t be delivered'}
1450                       %{Show Message.type#@Ip#'->'#Ipdest}
```

```
1451                        raise EnvoiExc end
1452                     end
1453                  end%catch
1454               end
1455
1456            end%local
1457
1458
1459         end%local
1460      end%method
1461
1462
1463
1464      meth recover_predecessor(step:S <= 1)
1465         skip
1466
1467         lock @PredTableLock
1468         then
1469            lock @PredListLock
1470            then
1471               %if S==1 then {Show 'recover predecessor called by'#@Id} end
1472               if S =< {List.length @PredList}
1473               then
1474                  try
1475                     {@Net.connect @Ip {List.nth @PredList S}.node.nodeIp}
1476                     {Array.put @PredTable 1 node(nodeId:{List.nth @PredList S}.node.nodeId
1477                                                  nodeIp:{List.nth @PredList S}.node.nodeIp)}
1478
1479                     %mise à jour de la liste des pred (laisser tomber les pred qui ne répondent plus)
1480                     PredList<-{List.drop @PredList S-1}
1481                  catch ErreurConnection
1482                  then
1483                     if ErreurConnection=='Already Connected'
1484                     then
1485                        {Array.put @PredTable 1 node(nodeId:{List.nth @PredList S}.node.nodeId
1486                                                     nodeIp:{List.nth @PredList S}.node.nodeIp)}
1487                        %mise à jour de la liste des pred (laisser tomber les pred qui ne répondent plus)
1488
1489                        PredList<-{List.drop @PredList S-1}
1490                     else
1491                        {self recover_predecessor(step:S+1)}
1492                     end%ErreurConnection=='Already Connected'
1493                  end%catch ErreurConnection
1494
1495
1496               else%if S=<List.length
1497                  {Array.put @PredTable 1 idle}
1498               end%if S<List.length
1499            end%lock
1500         end%lock
```

```
1501
1502        end%method recover_predecessor
1503
1504
1505        %Trouver le noeud X le plus proche du dernier noeud de la liste auquel on peut se connecter
1506        %Pré : Path == [A.....X....E]
1507        %res : [ThisNode X....E]
1508        meth shortestPath(path:Path res:ShortestPath step:Step <={List.length Path})
1509       %!!!!Ajouter un enlèvement de boucle.
1510           if Step==2
1511           then
1512              %!!!!Verifier : Path[1]=Noeud Courant
1513                  %Enlever boucle : {self removeCircuits(path:{List.append [node(id:@Id ip:@Ip)] {List.drop Path 1}}
1514                                                         res:ShortestPath)
1515              ShortestPath={List.append [node(id:@Id ip:@Ip)] {List.drop Path 1}}
1516           else
1517
1518           %Le chemin peut être court-circuité au niveau du noeud d'ip {{List.nth Path Step}.ip}
1519           %si il est connecté au noeud courant ou si une connexion noeud courant->noeud distant est possible
1520
1521              if {Bool.'or' {@Net.isConnectionPossible @Ip {List.nth Path Step}.ip}
1522                            {@Net.isConnected  {List.nth Path Step}.ip @Ip}}
1523              then
1524                 ShortestPath={List.append [node(id:@Id ip:@Ip)] {List.drop Path Step-1}}
1525              else
1526                 {self shortestPath(path:Path res:ShortestPath step:Step-1)}
1527              end
1528           end%Step==2
1529        end%   meth shortestPath(path:Path res:ShortestPath)
1530
1531        meth recover_successor(step:S <=1)%methode de résolution d'un successeur en échec
1532           lock @RecoverLock
1533           then
1534              local Succ
1535              in
1536                 %Successor tjrs idle ? (un autre thread n'a pas effectué le recovery ?)
1537                 {Array.get @FingerTable 1 Succ}
1538                 if Succ == idle
1539                 then
1540                    if S==1
1541                    then
1542                       skip
1543           %   {Show 'recover successor called by'#@Id#@SuccessorList}
1544                    end
1545                    if S =< {List.length @SuccessorList}
1546                    then
1547                       lock @FingerTableLock then
1548                          try
1549                             {@Net.connect @Ip {List.nth @SuccessorList S}.nodeIp}
1550
```

```
                        {Array.put @FingerTable 1 finger(node:node(key:@Id+1
                                                                   nodeId:{List.nth @SuccessorList S}.nodeId
                                                                   nodeIp:{List.nth @SuccessorList S}.nodeIp)
                                                         route:[node(id:@Id
                                                                     ip:@Ip)
                                                                node(id:{List.nth @SuccessorList S}.nodeId
                                                                     ip:{List.nth @SuccessorList S}.nodeIp)
                                                                ]
                                                         )
                        }
                        lock @SuccessorListLock
                        then
                  %mise à jour de la liste des successeur (laisser tomber les successeurs qui ne répondent plus
                           SuccessorList<-{List.drop @SuccessorList S-1}
                        end
                     catch ErreurConnection
                     then
                        if ErreurConnection=='Already Connected'
                        then
                           {Array.put @FingerTable 1 finger(node:node(key:@Id+1
                                                                      nodeId:{List.nth @SuccessorList S}.nodeId
                                                                      nodeIp:{List.nth @SuccessorList S}.nodeIp)
                                                            route:[node(id:@Id ip:@Ip)
                                                                   node(id:{List.nth @SuccessorList S}.nodeId
                                                                        ip:{List.nth @SuccessorList S}.nodeIp)]
                                                            )
                           }
                           lock @SuccessorListLock
                           then
                              SuccessorList<-{List.drop @SuccessorList S-1}
                           end
                        else
                           {self recover_successor(step:S+1)}
                        end%ErreurConnection=='Already Connected'
                     end%catch ErreurConnection
                  end%lock

               else%if S=<List.length
                  {self kill()}
                  skip
               end%if S<List.length
            end%if succ==idle
         end%local
      end%lock
   end%meth recover_successor

   meth kill()
      {self.Stabilize stop}
      {self.Notify stop}
      {self.BuildFingers stop}
```

```
1601            {self.CheckPred stop}
1602            {self.CheckConnections stop}
1603            {self.KeepAlive stop}
1604            {self.CheckSucc stop}
1605            %{@Net.delete @Ip}
1606            %{Show @Id#'Node Killed'}
1607         end%meth kill
1608
1609         meth registerConnection(ip:Ip)
1610            {Dictionary.put @Connections Ip {OS.time}}
1611         end
1612
1613
1614      end
1615
1616      proc{NewNode N}
1617         {New Node noop N}
1618      end
1619
1620   end
```

```
Seul le comportement du noeud à la réception d'un message
de type "reply_finger_find_successor" est différent par rapport à Nodeproxy


      meth readMessage(fromip:IP message:Message) %Analyse d'un message reçu

            []
               reply_finger_find_successor
            then
               thread
                  %Trouver l'index correspondant au message
                  local
                     Index_to_update
                  %!!!!Empêcher cette proc de boucler si cle invalide...
                     FindEntry=proc {$ Index Cle Res}
                                 if {Array.get @FingerIndexTable Index}==Cle
                                 then
                                    Res=Index
                                 else
                                    {FindEntry Index+1 Cle Index_to_update}
                                 end%Index==Cle
                              end
                  in
                     try
                        Index_to_update={FindEntry 1 Message.content.key}
                     %Remplacer l'entrée correspondante
                        lock @FingerTableLock then
                           if Message.content.key_succ_id==@Id %
                           then%Le noeud se situe virtuellement dans sa propre table de routage,
                                %insérer idle pour éviter le bouclage des transferts.
                              {Array.put @FingerTable Index_to_update idle}
                           else
                              local
                                 ShortestPath
                              in
                                 {self shortestPath(path:{List.reverse Message.path} res:ShortestPath)}
                                 %L'entrée est-elle directe ?
                                 if {List.length ShortestPath}==2
                                 then%Oui, enregistrer cette entrée dans la table de routage
                                       {Array.put @FingerTable Index_to_update finger(node:node(key:Message.content.key
                                                                                                nodeId:Message.content.key_succ_id
                                                                                                nodeIp:Message.content.key_succ_ip)
                                                                                      route:ShortestPath)}
                                 else%Non, version allégée ==> Ne pas considérer cette route
                                       {Array.put @FingerTable Index_to_update idle}
                                 end%if
                              end%local
                           end% Message.content.key_succ_id==@Id %
                        end%lock
                     catch _
```

```
51                      then
52                         skip
53                      end%try
54                   end%local
55                end
56             end%thread
57          end
```

```
1  Le code de NodeStandard est identique à celui de Nodeproxy,
2  Seul le code correspondant au calcul de la route la plus courte
3  change.
4        meth shortestPath(path:Path res:ShortestPath step:Step <={List.length Path})
5           %Pas de calcul de la meilleure route. Prise du chemin direct
6            ShortestPath={List.append [node(id:@Id ip:@Ip)] {List.drop Path {List.length Path}-1}}
7        end%   meth shortestPath(path:Path res:ShortestPath)
```

```
1  functor%Foncteur de construction de récolteur de statistiques
2  import
3     System(show:Show showInfo:ShowInfo)
4  export
5     newStats:NewStats %Procedure de creation d'instance de récolteur de statistiques.
6  define
7
8     class Stats
9
10       attr
11          Id %Identificateur de l'instance du Noeud
12          Ip %Adresse Ip de l'instance du Noeud
13
14          Net %Reseau physique sur lequel le noeud envoie et reçoit ses mesages
15          NetworkSize %Taille du réseau Chord
16
17
18          %NodesList = Union(KernelNodesList,ExternalNodesList)
19          NodesList %Liste des noeuds du système
20          KernelNodesList %Liste des noeuds du noyaux
21          ExternalNodesList %Liste des noeuds de périphérie
22
23
24          StatsLock %Verrou de modification des enregistrements statistiques.
25
26          NumberOfRequests %Nombre total de requêtes comptabilisées
27          NumberOfAnswers %Nombre total de réponses comptabilisées
28          NumberOfRightAnswers % Nombre total de réponses cohérentes (Le responsable renseigné est le responsable effectif de la clé)
29          NumberOfWrongAnswers % Nombre total de réponses incohérentes (Doit valoir <NumberOfAnswers> - <NumberOfRightAnswers>
30          TotalAnswerLength % Somme des longueurs des chemins empruntés par les requêtes ayant été résolues correctement.
31          %Dictionaire des requêtes
32          %Chaque entrée (clé = identifiant de noeud) de ce dictionaire est un dictionaire dont les clés sont les clés correspondant
33          %aux requêtes effectuées par le noeud
34          RequestDictionary
35
36
37       meth noop
38          skip
39       end
40
41       meth readMessage(fromip:IP message:Message) %Analyse d'un message reçu
42
43
44          thread
45             case {Record.label Message.type} of
46                request
47             then
48                lock @StatsLock then
49                   NumberOfRequests <- @NumberOfRequests + 1
50
```

```
51                {Dictionary.put {Dictionary.get @RequestDictionary Message.source.id} Message.content.key nil}
52             end
53          []
54             answer
55          then
56             lock @StatsLock then
57                NumberOfAnswers <- @NumberOfAnswers + 1
58
59                {Dictionary.put {Dictionary.get @RequestDictionary Message.source.id} Message.content.key Message}
60             end%lock
61          []
62             keep_alive
63          then
64             skip
65
66          else%case
67             {Show 'Unrecognized message'#@Id#@Ip}
68             {Show Message}
69             skip
70          end%case
71       end%thread
72    end%meth
73
74    %Initialisation du noeud
75
76    meth init(ip:IP id:ID network:Network
77              networkSize:NetworkS nodesList:NodesL kernelNodesList:KernelNodesL externalNodesList:ExternalNodesL)
78       {Show 'Initialising Stat'}
79
80       StatsLock <- {Lock.new}
81       NumberOfRequests <-0
82       NumberOfAnswers <-0
83       NumberOfRightAnswers <-0
84       NumberOfWrongAnswers <-0
85       NodesList<-NodesL
86       KernelNodesList<-KernelNodesL
87       ExternalNodesList<-ExternalNodesL
88       NetworkSize<-NetworkS
89       Net <- Network
90       Id<-ID
91       Ip<-IP
92
93
94       %Initialisation du dictionaire des requêtes
95       {Dictionary.new @RequestDictionary}
96       {Loop.'for' 1 {List.length @NodesList} 1
97        proc{$ NodeRank}
98           {Dictionary.put @RequestDictionary {List.nth @NodesList NodeRank} {Dictionary.new}}
99        end%proc
100      }%Loop
```

```
101
102
103            %Enregistrement auprès de la couche réseau
104            {@Net.insert self @Ip}
105            {Show 'Stat initialized'}
106         end
107
108         %Affichage des statistiques du noeud
109         meth showNode()
110            {Show 'Number of requests :'#@NumberOfRequests}
111            {Show 'Number of answers : '#@NumberOfAnswers}
112            %{Show {Dictionary.entries @RequestDictionary}}
113            %Calcul du pourcentage de réponse à une requête
114            local Pourcentage
115            in
116               Pourcentage ={Float.'/' {Int.toFloat @NumberOfAnswers}{Int.toFloat @NumberOfRequests}}
117               {Show 'Percentage of resolved requests : '#Pourcentage}
118            end%local
119
120            %Calcul du pourcentage de réponses cohérentes parmis les requêtes ayant reçu une réponse.
121            {Show 'Verifying Answers Correctness'}
122            NumberOfRightAnswers <- 0
123            NumberOfWrongAnswers <- 0
124            TotalAnswerLength <- 0
125            {Loop.'for' 1 {List.length @NodesList} 1
126             proc{$ NodeRank}
127                %{Show NodeRank#'Verifying Correctness of Answers received by'#{List.nth @NodesList NodeRank}}
128                local
129                   NodeRequests
130                   Keys
131                in
132
133
134                   {Dictionary.get @RequestDictionary {List.nth @NodesList NodeRank} NodeRequests}
135                   %NodeRequests contient les requêtes du <NodeRank>ème noeud de NodesList
136
137                   {Dictionary.keys NodeRequests Keys}
138                   %Keys est une liste contenant l'ensemble des clés
139                   %faisant l'objet d'une requête par le <NodeRank>ème noeud de NodesList
140
141   {Loop.'for' 1 {List.length Keys} 1
142    proc{$ Key}
143      local
144         AnswerMessage
145         Verif = proc {$ Step KeytoCheck AnswerMessage}
146             local
147                Answer = AnswerMessage.content.resp
148             in
149                if Step == 0-1
150                then skip
```

```
151               else
152                  if Step == 0
153                  then %Step == 0
154                  %{Show 'Verifying'#Step#KeytoCheck#Answer#{List.nth @NodesList 1}}
155 if {Bool.'or' {Value.'>' KeytoCheck {List.nth @NodesList {List.length @NodesList}}} {Value.'=<' KeytoCheck {List.nth @NodesList 1}}}
156                     then
157                  %La clé est plus grande ou plus petite que n'importe quelle valeur d'id du système
158                  %Le responsable de cette clé est obligatoirement le noeud ayant l'Id le plus petit du système
159                        if Answer == {List.nth @NodesList 1}
160                        then %Réponse cohérente
161                           lock @StatsLock
162                           then
163                              NumberOfRightAnswers <- @NumberOfRightAnswers + 1
164                              TotalAnswerLength <- @TotalAnswerLength +  AnswerMessage.pathlength
165                           end%lock
166                        else%réponse incorrecte
167                           skip
168                           lock @StatsLock
169                           then
170                              NumberOfWrongAnswers <- @NumberOfWrongAnswers + 1
171                           end%lock
172                          %{ShowInfo 'lookup for '#KeytoCheck#' initiated by '#{List.nth @NodesList NodeRank}#' returned
173                          %'#Answer#' by decision of '#AnswerMessage.realsource.id#'. Should have been '#{List.nth @NodesList 1}}
174                        end
175
176                     else%Clé comprise entre l'id le plus petit et l'id le plus grand du système
177                        {Verif 1 KeytoCheck AnswerMessage}
178                     end
179                  else%Step>0
180                  %{Show 'Verifying'#Step#KeytoCheck#Answer#{List.nth @NodesList Step+1}}
181                     if {Value.'=<' KeytoCheck {List.nth @NodesList Step+1}}
182                     then
183                  %le <Step+1>ème noeud de <NodesList> est le responsable effectif de la clé <KeytoCheck>
184                        if Answer == {List.nth @NodesList Step+1}
185                        then
186                           %lock @StatsLock
187                           %then
188                              NumberOfRightAnswers <- @NumberOfRightAnswers + 1
189                              TotalAnswerLength <- @TotalAnswerLength +  AnswerMessage.pathlength
190                           %end%lock
191                        else%réponse incorrecte
192                           %lock @StatsLock
193                           %then
194                              NumberOfWrongAnswers <- @NumberOfWrongAnswers + 1
195                           %end%lock
196                                   skip
197
198                        end%Réponse correcte
199                       %Sortie du test
200                        {Verif 0-1 KeytoCheck AnswerMessage}
```

```
                    else
                        %Test sur le suivant
                       {Verif Step+1 KeytoCheck AnswerMessage}
                    end%if
                 end%if step==0
              end%else
           end%local
        end%proc
     in
                        {Dictionary.get NodeRequests {List.nth Keys Key} AnswerMessage}
                        %Vérification de la validité de la réponse
                        try
                           local
                              Answer = AnswerMessage.content.resp
                           in
                              %Check de la validité de la réponse Answer par rapport à la <Key>ème clé de
                              %des clés sujettes à requêtes par le <NodeRank>ème noeud de <NodesList>
                              %{Show 'Checking validity of '#Answer#'as reply to request for key'#{List.nth Keys Key}}
                              {Verif 0 {List.nth Keys Key} AnswerMessage}
                           end%local
                        catch Except
                           %Exception possible : AnswerMessage vaut nil
                           %(pas de réponse à la requête) Impossible de vérifier la validité d'une réponse inexistante ==> Ignorer
                        then
                          % {Show 'Lookup for key:'#{List.nth Keys Key}#'initiated by'#{List.nth @NodesList NodeRank}#' failed...'}
                           skip
                           %{Show Except}
                        end
                     end%local
                  end%proc
               }
            end%local
         end%proc
        }%Loop


        %Affichage des statistiques.
        %{Show 'Number of requests :'#@NumberOfRequests}
        %{Show 'Number of answers : '#@NumberOfAnswers}
        {Show 'Number of Right answers : '#@NumberOfRightAnswers}
        {Show 'Number of Wrong answers : '#@NumberOfWrongAnswers}

        %{Show {Dictionary.entries @RequestDictionary}}
        %Calcul du pourcentage de réponse à une requête
        local Pourcentage
        in
           Pourcentage ={Float.'/' {Int.toFloat @NumberOfAnswers}{Int.toFloat @NumberOfRequests}}
           {Show 'Percentage of resolved requests : '#Pourcentage}
```

```
251          end%local
252
253          %Affichage du nombre de requêtes cohérentes par rapport au nombre de requêtes ayant été résolues
254          local Pourcentage
255          in
256             Pourcentage ={Float.'/' {Int.toFloat @NumberOfRightAnswers}{Int.toFloat @NumberOfAnswers}}
257             {Show 'Percentage of right resolved requests (% NumberOfAnswers) : '#Pourcentage}
258          end%local
259          local Pourcentage
260          in
261             Pourcentage ={Float.'/' {Int.toFloat @NumberOfRightAnswers}{Int.toFloat @NumberOfRequests}}
262             {Show 'Percentage of resolved and right resolved requests (% NumberOfRequests) : '#Pourcentage}
263          end%local
264          local LongueurMoyenne
265          in
266             LongueurMoyenne={Float.'/' {Int.toFloat @TotalAnswerLength}{Int.toFloat @NumberOfRightAnswers}}
267             {Show 'Average right answer path length : '#LongueurMoyenne}
268          end%local
269
270       end%meth
271
272       meth getSucc(succ:Succ)%Retourne l'ID du successeur (-1 si pas de successeur courant)
273          Succ = ~1
274       end
275
276
277       meth eqbefore(low:L high:H place:P <= @Id res:R)
278          LAux = {Int.'mod' L + @NetworkSize - P @NetworkSize}
279          HAux = {Int.'mod' H + @NetworkSize - P @NetworkSize}
280       in
281          R = LAux =< HAux
282       end
283
284       meth kill()
285          skip
286       end%meth kill
287
288       meth getDictionaryEntries(requetes:Dec)
289          Dec = {Dictionary.entries @RequestDictionary}
290       end%meth
291
292    end
293
294    proc{NewStats N}
295       {New Stats noop N}
296    end
297 end
298
```

```
1  functor
2  import
3  %    System
4     System(show:Show)
5     Module
6     OS
7     Application
8  %    Module
9  define
10
11
12 local
13    Network
14    Debug
15    StatHarvester
16    Nodes
17
18    NumberOfSimulations = 1
19    NetworkSize = 2000
20    NumberOfNodes = 100
21    FingerTableSize = 11
22    SuccessorListSize = 11
23 %    SuccessorListSize
24    KernelSize = 84
25    ExternalSize
26    LookupRange = 5
27    GenerateNumber
28    GenerateNumberList
29    NodeLookupDelay = 1000
30    KeyLookupDelay = 0
31 in
32 %Creation du réseau
33 [Network Debug] = {Module.link ["./Network.ozf"
34                                 "x-oz://boot/Debug"
35                                ]}
36 [StatHarvester] = {Module.link ["./Stats.ozf"
37                                ]}
38 %Choix du type de noeud
39 %NodeStandard, Nodeproxy ou  NodeproxyLite
40 [Nodes] = {Module.link ["./NodeproxyLite.ozf"
41                        ]}
42
43
44 %Initialisation du générateur aléatoire de nombres entiers
45 {OS.srand 0}
46
47 ExternalSize = NumberOfNodes - KernelSize %Nombre de noeuds exterieurs au noyau
48
49 {Network.initMatrix NetworkSize} %Initialisation de la matrice d'accessibilité
50
```

```
51
52  %Ecart inter-clé pour l'injection des requêtes.
53  %Si LookupRange = 1 : Toutes les clés sont testées.
54  %Si LookupRange = 2 : les clé 1 3 5 7 9 ... sont testées.... etc.
55  %LookupRange = 50
56
57  %Procédure de génération aléatoire d'Id de noeud.
58  GenerateNumber = proc {$ Res}
59                      local
60                         RandomNumber
61                      in
62                         {Int.'mod' {OS.rand} NetworkSize RandomNumber}
63                         Res = RandomNumber +1
64                      end%local
65                   end%proc
66  %Procédure de génération des noeuds du système
67  GenerateNumberList = proc {$ TList From To}
68              if {Value.'>' From To}
69              then
70                 skip
71              else
72                 local NewNode = {GenerateNumber}
73                 in
74                   % {Show 'Trying'#NewNode}
75                    if {List.member NewNode {List.take TList From-1}}
76                    then
77                     %  {Show NewNode#'Already assigned'}
78                       {GenerateNumberList TList From To}
79                    else
80                       {List.nth TList From} = NewNode
81                    %   {Show 'Node inserted, Id :'#NewNode}
82                       {GenerateNumberList TList From+1 To}
83                    end%test de pré-existence d'un noeud d'ID NewNode
84                 end%local
85              end%if Nodeid == Array.get(From)
86           end%proc
87
88
89  {Loop.'for' 1 NumberOfSimulations 1
90   proc{$ SimNumber}
91      local
92         %Création des listes de noeuds
93         NodesList = {List.make NumberOfNodes}
94         KernelNodesList = {List.make KernelSize}
95         ExternalNodesList = {List.make ExternalSize}
96         TempList
97         Stat
98         FirstNode
99         StabInfoThread
100     in
```

```
101         {Show '*************************************************************************'}
102         {Show '************Starting simulation number'#SimNumber#'**************************'}
103         {Show '*************************************************************************'}
104
105
106
107
108         %Prendre les derniers générés comme les externes. Créer les deux listes  correspondantes. Trier les listes.
109         TempList = {List.make NumberOfNodes}
110         {Show 'Generating the nodes....'}
111         {GenerateNumberList TempList 1 NumberOfNodes}
112         {Show 'Finished....'#NumberOfNodes#'random Id numbers created.'}
113         {Show 'Taking '#KernelSize#'of them as Kernel Nodes'}
114         {List.sort {List.take TempList KernelSize} Value.'<' KernelNodesList}
115         {Show 'Taking '#ExternalSize#'last ones as External Nodes'}
116         {List.sort {List.drop TempList KernelSize} Value.'<' ExternalNodesList}
117         {List.sort TempList Value.'<' NodesList}
118         {Show 'Finished'}
119
120
121         %Initialisation des noeuds et du système.
122         %Initialisation du récolteur de statistiques
123         {Show 'Initialising Statistic Harvester'}
124         Stat = {StatHarvester.newStats}
125         {Stat init(ip:0-1 id:0-1
126                    network:Network networkSize:NetworkSize
127                    nodesList:NodesList kernelNodesList:KernelNodesList
128                    externalNodesList:ExternalNodesList)}
129         {Show 'Finished'}
130
131
132         %Pour l'organisation des contacts. Seul les noeuds du noyaux sont des contacts d'entrée dans le système.
133         %Pour assurer la création d'un seul anneau Chord,
134         %  tous les noeuds du noyau doivent avoir un contact d'entrée d'identifiant inférieur au noeud entrant.
135         %Le noeud du noyau ayant l'identifiant le plus faible est le premier noeud du système. Il ne contacte aucun noeud.
136
137         {Show 'Initalising Network Topology...'}
138         %Initialisation de la topologie du réseau
139         %Tous les noeuds du noyau sont accessibles
140         {Loop.'for' 1 KernelSize 1
141          proc{$ ToCounter}
142             {Loop.'for' 1 NetworkSize 1
143              proc{$ FromCounter}
144                 {Network.setConnectionPossible FromCounter {List.nth KernelNodesList ToCounter} true }
145              end
146             }
147          end
148         }
149         {Show 'Access to Kernel Nodes granted'}
150
```

```
151         %Aucun noeud de la périphérie est joignable
152         {Loop.'for' 1 ExternalSize 1
153          proc{$ ToCounter}
154             {Loop.'for' 1 NetworkSize 1
155              proc{$ FromCounter}
156                 {Network.setConnectionPossible FromCounter {List.nth ExternalNodesList ToCounter } false}
157              end
158             }
159          end
160         }
161         {Show 'Access to External Nodes blocked'}
162
163
164         %Initialisation du premier noeud
165         FirstNode = {Nodes.newNode}
166         {FirstNode init(ip:{List.nth KernelNodesList 1}
167                         id:{List.nth KernelNodesList 1}
168                         network:Network
169                         networkSize:NetworkSize
170                         fingerTableSize:FingerTableSize
171                         successorListSize:SuccessorListSize
172                         predTableSize:1 )}
173
174
175
176
177         %Entrée des noeuds du noyau.
178         {Show 'Kernel Nodes are joining the ring '}
179         {Loop.'for' 2 KernelSize 1
180          proc{$ Counter}
181             local
182                ContactNode
183                TempNode = {Int.'mod' {OS.rand} Counter} + 1
184                NewNode = {Nodes.newNode}
185             in
186                if TempNode == Counter
187                then
188                   ContactNode = TempNode- 1
189                else
190                   ContactNode = TempNode
191                end
192
193                {NewNode init(ip:{List.nth KernelNodesList Counter}
194                              id:{List.nth KernelNodesList Counter}
195                              network:Network
196                              networkSize:NetworkSize
197                              fingerTableSize:FingerTableSize
198                              successorListSize:SuccessorListSize
199                              predTableSize:1 )}
200                {NewNode join(contactId:{List.nth KernelNodesList ContactNode}
```

```
                                contactIp:{List.nth KernelNodesList ContactNode})}
                {Time.delay 200}
             end%local
          end
         }
         {Show 'Kernel Nodes In'}

         {Show 'Waiting'}
         {Time.delay 5000}



         %Entrée des noeuds externes
         {Show 'External Nodes are joining the ring'}

         {Loop.'for' 1 ExternalSize 1
          proc{$ Counter}
             local
                ContactNode = {Int.'mod' {OS.rand} KernelSize} + 1
                NewNode = {Nodes.newNode}
             in
                {Show 'External node :'#{List.nth ExternalNodesList Counter}}
                {NewNode init(ip:{List.nth ExternalNodesList Counter}
                              id:{List.nth ExternalNodesList Counter}
                              network:Network
                              networkSize:NetworkSize
                              fingerTableSize:FingerTableSize successorListSize:3 predTableSize:1 )}
                {NewNode join(contactId:{List.nth KernelNodesList ContactNode}
                              contactIp:{List.nth KernelNodesList ContactNode})}
                {Time.delay 75}
             end%local
          end
         }
         {Show 'External Nodes In'}


%Thread d'affichage de l'évolution des stabilisations
StabInfoThread = {New Time.repeat
                  setRepAll(action: proc {$}
                                       local
                                          MaxStabilized = {Array.new 1 3 0}
                                       in
                                          {Array.put MaxStabilized 2 0}
                                          {Array.put MaxStabilized 2 0}
                                          {Loop.'for' 1 NumberOfNodes 1
                                           proc{$ Counter}
                                              local
                                                 Idem
                                                 Successor = {Network.getMachineSucc {List.nth NodesList Counter}}
                                              in
```

```
                                                if Counter==NumberOfNodes
                                                then
                                                   Idem = Successor=={List.nth NodesList 1}
                                                   if Idem
                                                   then
                                                      {Array.put MaxStabilized 2 {Array.get MaxStabilized 2}+1 }
                                                   else
                                                      skip
                                                   end
                                                else
                                                   Idem = Successor=={List.nth NodesList Counter+1}
                                                   if Idem
                                                   then
                                                      {Array.put MaxStabilized 2 {Array.get MaxStabilized 2}+1 }
                                                   else
                                                      skip
                                                   end
                                                end
                                          end%local
                                               end%proc
                                              }
                                              {Show 'Number of Stabilized nodes : '#{Array.get MaxStabilized 2}}
                                           end%local
                                        end%proc
                        delay: 60000
                        number: ~1
                              )
                     }

   %Lancement du thread informant de la stabilité du réseau
   thread {StabInfoThread go} end

   {Show 'Waiting'}
   {Time.delay 15000}
   %Envoi requêtes de type lookup
   {Show 'Injecting lookups with inter key range'#LookupRange}
   {Loop.'for' 1 {List.length NodesList} 1
    proc{$ NodeRank}
       {Time.delay NodeLookupDelay}
        if {Int.'mod' NodeRank 20}==10
            then
            {Show NodeRank#'Sending lookups from node'#{List.nth NodesList NodeRank}}
            end
       {Loop.'for' 1 NetworkSize LookupRange
        proc{$ C}
           local
              KEY = {Int.'mod' {List.nth NodesList NodeRank}+C NetworkSize-1}+1
              LookuptoSend = message(type:lookup
                                     source:node(id:{List.nth NodesList NodeRank} ip:{List.nth NodesList NodeRank})
                                     path:[node(id:{List.nth NodesList NodeRank} ip:{List.nth NodesList NodeRank})]
```

```
                                        content:content(key:KEY))
                  StattoSend = message(type:request
                                       source:node(id:{List.nth NodesList NodeRank} ip:{List.nth NodesList NodeRank})
                                       path:{List.make 0}
                                       content:content(key:KEY))
               in
                  {Network.feedMessage {List.nth NodesList NodeRank} LookuptoSend}
                  {Network.sendStat 0-1 StattoSend}
                  {Time.delay KeyLookupDelay}
               end%local
            end%proc
           }%loop
         end%proc
        }%loop
        {Show 'All lookups are sent waiting for all answers to arrive'}

        %Temps d'attente : Pour être sûr que toutes les requêtes
        %aient le temps de voir leur réponse arriver (si l'environnement permer de trouver une réponse) à leur destinataire
       {StabInfoThread stop}
       {Show 'Stabilization computing stopped'}


       {Time.delay 30000}

        %Constitution et affichage des statistiques collectées.
        {Stat showNode()}

        %Tuer tous les noeuds, "nettoyage" du réseau avant la simulation suivante
        {Show 'Cleaning the network'}
        {Loop.'for' 1 NumberOfNodes 1
         proc{$ NodeRank}
            {Network.delete {List.nth NodesList NodeRank}}
         end%proc
        }%loop
        %Suppression du noeud récolteur de statistiques
        {Network.delete ~1}
        {Show 'Network Cleaned'}
        %{Time.delay 20000}
     end%local
  end%proc(boucle sur les simulations)
  }%loop
 {Show 'All simulations done'}

 {Application.exit 0}
 end
 end
```